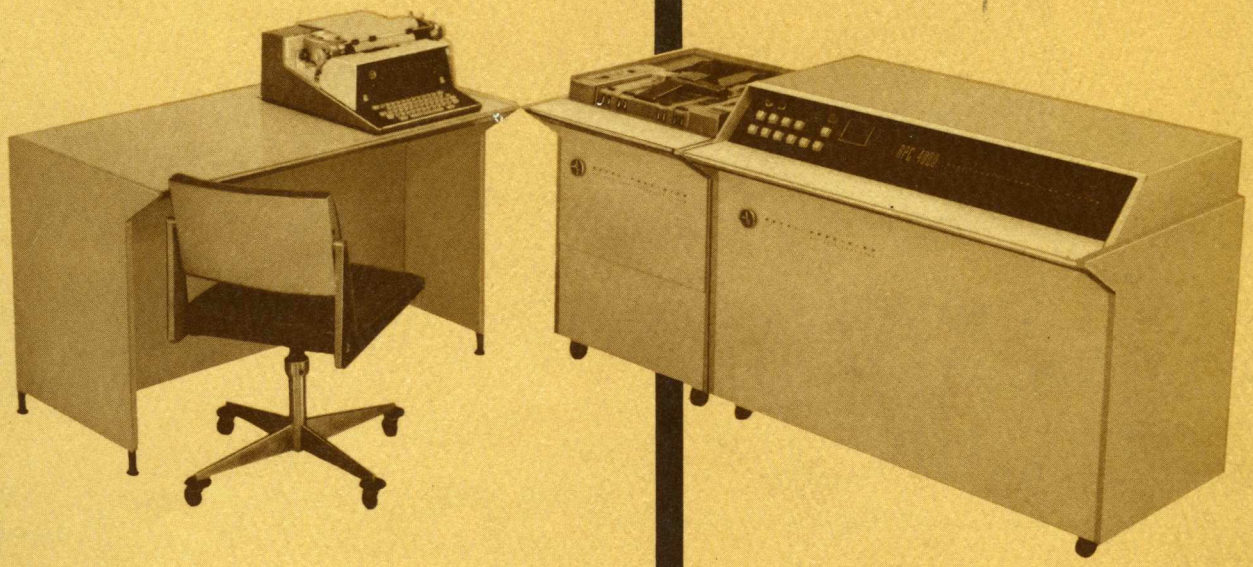
RPC 4000

H3-02.0



ACT IV programming manual

GENERAL PRECISION, INC.
Commercial Computer Division

ACTIV

AN ALGEBRAIC COMPILER
AND TRANSLATOR

for the **RPC 4000** General Precision Electronic Computer

PROGRAM NO. H3-02.0

INTRODUCTION

ACT IV is an algebraic compiler and translator which was specifically developed for General Precision's RPC-4000 computer. In this manual, ACT IV refers to the compiler as well as to the language which it interprets. Users of General Precision's LGP-30 computer will recognize ACT IV as a descendant of that computer's ACT III language. Like it, it is not far removed from the familiar language of mathematics.

The ACT IV processor—as distinguished from the language—is an RPC-4000 program supplied by General Precision, Inc. It translates problem statements from ACT IV language into RPC-4000 machine language and stores the translation in the computer for immediate execution. The processor includes the arithmetic and other subroutines required for running the translated program. Thus, one could use the RPC-4000 by means of this processor without any knowledge of machine language programming.

SCOPE OF MANUAL

Part 1 of this manual defines the vocabulary and rules of the ACT IV language.

Part 2 gives the operating instructions for the ACT IV processor, including methods for locating and correcting programming errors.

Part 3 contains a number of ACT IV sample programs. Appendix A briefly explains the internal number representation for the RPC-4000, and Appendix B contains a collection of reference lists and tables.

In addition to this manual, a separate ACT IV Reference Manual is available from General Precision, Inc. It covers technical and operational aspects of ACT IV which do not belong in the programming manual proper. Also available separately is a ROAR symbolic tape of the processor.

CONTENTS

THE ACT IV LANGUAGE 1

STATEMENTS

The main body of an ACT IV program consists of a list of statements. A statement can do one or more of the following:

Call for numerical or other data to be entered into the computer from tape or typewriter.

Evaluate mathematical formulas.

Call for the execution of procedures defined earlier within the ACT IV program.

Print or punch the results of calculations in any desired format. Alphabetic information can be included in the output.

Modify the normal sequence of statement execution. For example, a statement may cause the repetition of groups of statements a specified number of times.

VARIABLES

Two types of numerical quantities may occur in the calculations in an ACT IV program: constants, which are explicitly written out in the program, or variables, which are either entered from tape or typewriter during a computation, or obtained through evaluation by a formula in the course of the program.

In mathematics, variables are usually referred to by single letters. In ACT IV, on the other hand, a variable may be given any name consisting of up to 5 characters, as long as the name is not reserved for special purposes in the ACT IV system. Restrictions on naming conventions may be summarized as follows:

Names of ACT IV operations may not be used as names of variables. (A complete list of ACT IV operations appears in Table 3 of Appendix B of this manual.)

Names of variables may not consist entirely of digits, periods, spaces, plus- and minus-signs, as they could be mistaken for numerical constants.

Names of variables may not end with two periods. Such names could be mistaken for statement labels by the processor.

The asterisk, (*), can not be one of the characters of a name.

Of these restrictions, only the first is likely to require careful attention. In particular, "x" is the symbol for the multiplication operation, thus it can not be used by itself as the name of a variable. One might use "ex", "xx", or of course "y" instead.

Note that the typewriter space is a character, and must not be inserted without reason. For example, "asq", " asq", "asq ", and "a sq" would be treated as four different variables by the processor.

The computer does not distinguish between upper and lower case symbols in input. Therefore "n" and "N" are considered as the same name, as are "mass" and "MASS", "Σy" and "4y", and so on. Further, the question mark, (?), is not allowed as a variable name, since it could not be distinguished from the operator (+), as the two characters represent the upper and lower case symbols of the same key on the RPC-4000 typewriter keyboard.

FLOATING-POINT NUMBERS

Numbers in the ACT IV system can be handled in fixed-point or in floating-point form. In most programs, calculations in floating-point predominate.

The exact internal representation of numbers in these two forms will be found in Appendix A of this manual. At this point only information of direct use to the programmer is presented.

A floating-point number can be positive or negative, and can have magnitudes in the range

from .5 x 2^{-128} (which is about 1.5×10^{-39})
to .999... x 2^{+127} (which is about $1.7 \times 10^{+38}$),
as well as zero.

Floating-point operations with whole numbers (0, ±1.0, ±2.0, etc.) are exact as long as the numbers and results range below 16 million. Most calculations, of course, are not with whole numbers. In general, the computer retains between 7 and 8 significant decimal digits (24 binary digits) of accuracy when numbers are first introduced in a calculation. A series of arithmetic operations on floating-point numbers may yield a result with greater error. This may be due to cumulative conversion errors or to rounding-off processes which take place after many of the operations. Such errors are unavoidable, but usually small. While floating-point arithmetic is somewhat slower than fixed-point arithmetic, it is the preferred form for most scientific and engineering computations, because of the broad range allowed for floating-point numbers.

The necessary machine-language subroutines for operations with floating-point numbers on the RPC-4000 are included in the ACT IV processor.

FLOATING-POINT CONSTANTS

Computations frequently involve constants; that is, numbers whose values remain unchanged throughout the same program. If a formula contains a constant which is to be used in floating-point form, this is indicated by typing the number with a decimal point (.) in the proper place. Negative constants are indicated by a minus sign (-), normally preceding the number. For a positive constant, the plus sign (+) can be used for emphasis, but is not required. Spaces may be typed within a number, or before or after it, if this aids legibility. In this instance, they are ignored by the processor. (This is not the case elsewhere in the ACT IV language.)

The expression for a constant must not contain more than 10 characters, including decimal point, sign, and spaces, if any. If more characters are used, only the last 10 are examined by the processor.

A floating-point constant must include a decimal point. Special case: Zero can be written as "0", without a decimal point.

Examples:

1.0 +1. + 1.00 1.+	= Plus One
-3.1416 3.1416- -3.1 4 1 6	= Minus Pi

THE STOP CODE(*)

When typing or punching an ACT IV program, every variable name, constant, operation symbol, and command symbol must be followed by the asterisk, that is, a stop code, (*).

Compare the two expressions

a*+*b*/*c* and a*+*b/c*

The first expression contains three variables (A, B, C) and two operations (+ and /). The second contains only two variables (A and B/C) and one operation (+), since B/C is taken as a single three-character name, unrelated to the possible variables B and C.

FLOATING-POINT OPERATIONS

BASIC OPERATIONS

/ x - +

are the symbols used to denote the four familiar arithmetic operations, when they are to act on floating-point quantities.

The sign (-) represents only subtraction, not sign-change. Thus the form

-*u*-*v*

is illegal, if the first (-) specifies a sign-change, not the subtraction of U from something. (See the "minus" operation described below.)

In algebra, the (x) symbol for multiplication is often omitted, or replaced by " " In ACT IV "x" must be used; juxtaposition does not suffice to indicate multiplication.

SIGN OPERATIONS

minus

denotes sign-change for floating-point numbers.

Consider -*u*-*v* again. Since - u - v is mathematically the same as 0 - u - v, it could be written as

0*-*u*-*v*

a correct ACT IV expression. However, at a saving of both translating and computing time,

minus*u*-*v*

can be used.

abs

denotes the operation of absolute value, that is, making the sign plus, for floating-point numbers.

Thus abs*v*

yields V if V is already positive, and V with sign changed (0 - v) if V is negative.

OTHER FLOATING-POINT OPERATIONS AND FUNCTIONS

pwr

a*pwr*b* yields A raised to the power B, (A^B). Both A and B must be floating-point numbers; and A must be positive. (The subroutine for pwr first calculates $\log_e a$, and the logarithm function is not defined for negative numbers.)

If B is a small positive whole-number constant, pwr is slower and less accurate than repeated multiplication: for example, use a*x*a*x*a*, not a*pwr*3.*.

sqrt

sqrt*a* yields the square root of A; A must not be negative.

ln

ln*a* yields the natural, or base e, logarithm of A; A must be positive. For the base 10 logarithm, use ln*a*/*2.3025851*.

exp

exp*a* yields the exponential of A, e^A. For 10^A, use exp*[*a*x*2.3025851*]*.

sin

sin*a* yields the sine of the angle A; A must be expressed in radians. If A is in degrees, use sin*[*a*/*57.29578*]*.

cos

cos*a* yields the cosine of the angle A; A must be expressed in radians. If A is in degrees, use cos*[*a*/*57.29578*]*.

artan

artan*a* yields the arctangent of the number A. The answer is in radians, $-\pi/2$ to $+\pi/2$. If the answer is wanted in degrees, use [*artan*a*]*x*57.29578*.

tanh

tanh*a* yields the hyperbolic tangent of the number A.

PRECEDENCE AND BRACKETS

The ACT IV processor groups algebraic operations according to the usual precedence conventions of algebra. In the absence of brackets, the sign operations, pwr, and functions (ln, sin, and so on) are evaluated first, from left to right; next, multiplications and divisions; finally, additions and subtractions.

For example, (dropping the (*) codes for clarity),

a + b x c pwr d / e - sin f + g

is interpreted as

$$a + \frac{b.\ c^d}{e} - (\sin f) + g$$

[ ]

As in algebra, the precedence of operations can be modified by the use of brackets. Only the square brackets on the RPC-4000 typewriter may be used, not the parentheses which appear as upper case symbols on the "9" and "0" keys. The precedence rules described above are first applied within the innermost set of brackets, yielding a numerical result which is then used in evaluating the next set of brackets, and so on, working from inside out. In other words, the part or parts enclosed by the greatest number of brackets are evaluated first, as the statement is scanned from left to right; then the next most "deeply" enclosed parts, from left to right; and so on, until no further brackets are left.

For example,

a pwr b pwr c

is interpreted as $(a^b)^c$, since a pwr b is evaluated first. For greater clarity, this might be written as

[a pwr b] pwr c

or better, since $(a^b)^c = a^{bc}$, as

a pwr [b x c]

On the other hand, if the term, above, is meant to signify $a^{(b^c)}$, then it should be written as

a pwr [b pwr c]

As a last example,

$$\frac{-b + \sqrt{b^2-4ac}}{2a}$$

would be written—this time with the proper (*) codes—as

[*minus*b*+*sqrt*[*b*x*b*-*4. *x*a*x*c*]*]*/*[*2. *x*a*]*

Notice that the argument of sqrt, b^2-4ac, has to be bracketed, otherwise

sqrt*b*x*b*-*...

would be taken as

$\sqrt{b}$ x b - ...

The numerator is in brackets, or only the square root would be divided; and the denominator is in brackets, for otherwise the result would be

$$\frac{-b + \sqrt{b^2-4ac}}{2} \times a$$

Note, the end of the expression could be written

```
...]*/*2. */*a*
```

since the divisions are done in sequence from left to right. However, this arrangement is not recommended, since it would be unclear to most readers of the program.

In an expression, brackets may not be nested more than 14 deep. In the rare event when this limit needs to be exceeded, two statements should be used; one to evaluate the innermost few brackets and assign the result some name, such as "temp"; and the second using "temp" in place of these brackets.

ASSIGNMENT STATEMENTS

=

The equal sign is used in ACT IV to denote assignment or substitution. For example,

```
sqrt*[*a*x*a*+*b*x*b*]*=*rms**
```

is a statement which calculates $\sqrt{a^2 + b^2}$ (assuming that the variables A and B have received numerical values earlier in the program), and assigns that value to the variable RMS.

If a value had been assigned to RMS earlier in the program, the assignment statement replaces that value, and the former value is discarded. As another example of this, the statement

```
path*+*leg*=*path**
```

will take the previous value of the variable PATH, add the value of the variable LEG to it, and change the value of PATH to this new value.

The statement `1. 0*=*ex**` assigns the value 1 in floating-point form to the variable EX. Notice that the substitution is from left to right: `EX*=*1. 0**` is incorrect. It may assist the programmer to read the (=) symbol as "yields" or "replaces."

(blank)*

The reader may have noticed that each of the assignment statement examples just given ended with two (*) codes. The first is the stop code which is typed after every name. The second (*), which follows immediately (and is described as "blank"), marks the end of the statement. Every statement in ACT IV, not only assignment statements, must end with a second (*).

MULTIPLE ASSIGNMENTS

If the same value is to be given to several variables, one statement with several (=) signs may be used. For example,

```
a*+*. 001*=*a*=*red**
```

assigns the value of the sum: former value of A, plus . 001, to both the variables A and RED.

Multiple assignments are very commonly used to set several variables to zero as an initial value.

```
0*=*sumx*=*sumy*=*count**
```

sets all three variables—SUMX, SUMY, and COUNT—to zero.

FIXED-POINT NUMBERS, CONSTANTS, AND OPERATIONS

NUMBERS AND CONSTANTS

In ACT IV fixed-point (or "integer") numbers are used primarily for counting and as subscripts. A fixed-point number can be zero or any positive or negative integer less in magnitude than

$$2,147,483,648 \text{ or } (2^{31})$$

In the usual applications, integers will be much smaller than this upper limit.

A constant which is to be used in fixed-point form in a formula, is typed as a number without a decimal point. In all other respects, the form is the same as for floating-point constants. Notice that the constant zero can be written in the same way whether it is to be used as a fixed- or floating-point number.

BASIC OPERATIONS

i+ i- ix i/

are the symbols which denote the four familiar arithmetic operations when they are performed on fixed-point quantities. The sign (i-) represents only subtraction, not sign-change. No fixed-point operation analogous to the floating-point "minus" is provided in ACT IV. If U and V are variables whose values are in fixed-point form, then - u - v can be expressed in the form `0*i-*u*i-*v*` .

The (i/) operation requires further explanation. If A is not exactly divisible by B, and B is positive, the answer obtained from a*i/*b* will be the largest integer contained in the true quotient A/B. Hence it will be less then the true quotient. Thus

```
 104*i/*-5*    yields    +20  (less than +20. 8);
-104*i/*-5*    yields    -21  (less than -20. 8).
```

If B is negative, the answer will be greater by a fraction than the true quotient. Thus

```
 104*i/*-5*    yields    -20  (greater than -20. 8);
-104*i/*-5*    yields    +21  (greater than +20. 8).
```

OTHER FIXED-POINT OPERATIONS, BRACKETS, AND ASSIGNMENTS

rmain

If the denominator of an (i/) operation is followed immediately by, for example, rmain*r*, then R is assigned the value of the remainder of the division, in fixed-point form. Thus the statement

```
[*f*i+*g*]*i/*[*m*i+*n*]*rmain*r*=*q**
```

assigns to Q the quotient of the division

$$\frac{f+g}{m+n},$$

and assigns to R the remainder of the division.

In any case, the remainder is always positive (or zero), and is related to the quotient Q given by the (i/) operation by the formula

$$\frac{a}{b} = q + \frac{r}{b}$$

iabs

denotes the operation of absolute value, that is, making the sign plus, for fixed-point numbers.

[ ]

The rules for precedence and bracketing are the same for expressions calculated in fixed- as in floating-point form.

=

The equal sign can be used for assignments in fixed- as well as floating-point form. For example, if U and V are variables whose values are in fixed-point form, then the statement

```
2*ix*[*u*i+*v*]*=*k**
```

assigns to the variable K the value 2(u + v) in fixed-point form.

FORM CONVERSION

0*flo*

If B is a variable whose value is in fixed-point form, 0*flo*b* yields the value of B, converted to floating-point form. Thus the statement 0*flo*b*=*blq** gives to the variable BLQ the value of B, but in floating-point form. "0*flo*" is the most commonly used case of the more general operation defined in the next paragraph.

flo

If A and B are both in fixed-point form, a*flo*b* yields the value $b/10^a$ in floating-point form. For example, if the variable CENTS represents a number of cents in fixed-point form, then 2*flo*cents* yields the number of dollars in floating-point form. The value of the integer A, in general, represents the number of positions of shift desired for the decimal point.

unflo

If B is in floating-point. unflo*b* yields the value of b rounded to the nearest whole number and converted to fixed-point form.

fix

If B is in floating-point, fix*b* yields the truncated value of B converted to fixed-point form.

INPUT

OPERATIONS

read

read*a* causes one number to be read from whichever input device had been previously selected, and assigns that value to the variable A in floating-point form.

iread

iread*a* is the same, except that the value assigned to A is in fixed-point form.

aread

aread*a* causes one alphanumeric word of five or fewer characters to be read from whichever input device had been previously selected, and stores that word as the value of the variable A in a form suitable for later printing by "aprt".

rdhex

rdhex*a* reads a hexadecimal word of 8 or fewer characters and stores it as the value of the variable A. (This operation is used primarily in conjunction with "hxpch" to be defined later.)

At this point, it might be remembered that, when the RPC-4000 executes a program translated from the ACT IV language, it executes statement after statement, from top to bottom, unless the sequence is varied by transfer operations written explicitly into the program. Thus, if several statements in the program contain input operations—perhaps with transfer operations causing some or all of them to be repeated one or more times—these input operations will be executed in a definite sequence of which the programmer must be aware. If the input device which has been selected (manually or under program control) is a tape reader, the data on the tape must have been punched in the sequence in which it will be called for. If the selected input device is the typewriter, the typist must supply the correct data at the correct time. In this case, it is suggested that the program print out an identifying heading before each call for input.

FORMAT FOR DATA INPUT

read

A number to be read and stored in floating-point form can be typed or punched on tape in the same form as indicated for floating-point constants within the ACT IV program. That is, it appears as a number with a decimal point appearing in the appropriate position, a (-) sign for negative numbers, and is followed by an asterisk. (A (+) can be used for positive numbers, but is not required.) In addition, floating-point notation may be used. A floating-point number is written as follows:

a. A number containing not more than 9 digits, preceded by its sign if negative.
b. A decimal point, if desired. (If no decimal point is entered, it is assumed to follow the last digit of the number.)

c. The letter "E" to introduce the exponent (i. e. , the power of 10 by which the number must be multiplied to produce the value desired).
d. The value of the exponent and its sign, if negative.

For example:

to input	1234567800
type	12345678E2*
or	.12345678E+10*
to input	.000012345678
type	.12345678E-4*
or	12345678E-12*

A (-) appearing anywhere before the "E" is interpreted as the sign of the number; after the "E", as the sign of the exponent.

Note: The forms containing the letter "E", described here, can not be used to express floating-point constants within ACT IV programs.

Any data input by a "read" operation is automatically considered floating-point. Thus 1* is sufficient for floating-point input of the number one; 1E6* for the number one million.

There may not be more than 9 digits in the number typed in, excluding the exponent, if any. To increase legibility, spaces may be typed anywhere within, before or after a number. The entire expression must not contain more than 16 characters, however, counting spaces. (If more than 16 are used, only the last 16 are considered by the read routine in the processor.)

iread

A number to be input by "iread" should be typed simply as the number itself, with a (-) if negative, followed by a (*) code. (+) for positive numbers is optional. Spaces may be included for legibility. The expression must not contain more than 16 characters.

A decimal point in a number input by an "iread" operation is ignored. Thus, if a dollars-and-cents amount is typed in with a decimal point when an "iread" operation is calling for input, the total number of cents will be stored in fixed-point form.

aread

The "aread" operation inputs and stores (right justified) any group of 5 or fewer acceptable characters, followed by (*). Among these characters can be spaces, but not (*) codes. Other format controls, such as carriage return, tab, upper case, lower case and back space, are ignored. Whatever characters are read in and stored using "aread" can subsequently be printed out using "aprt". If names, headings, etc. of more than 5 characters are to be read in and later printed out, a series of "aread" operations should be used which store each five-or-fewer character group as a different variable. Note that, in preparing the input, a stop code must appear every 5 or fewer characters, and the number of asterisks must be the same as the number of "aread" operations used. If the input for an "aread" operation is blank (that is, a sequence of two (*) codes) nothing will be printed by the corresponding "aprt" operation.

rdhex

The input for a "rdhex" operation—or series of "rdhex" operations—will usually be a tape produced earlier by a "hxpch" operation. It will consist of a word of 8 or fewer hexadecimal characters (0 through 9, A through F) followed by (*). When there are fewer than 8 characters, leading zeros are automatically supplied. For an explanation of a use of this operation, see "hxpch".

OUTPUT

GENERAL

The results of the computer's calculations may be printed out in a variety of formats which must be specified by the programmer. First of all, the appropriate output device(s) must be selected. This determines whether the results are typed out, punched on tape, or both. If the results on tape are intended for subsequent off-line printout, the same considerations of output layout apply as for typed copy. Data may also be punched on tape for later re-entry into the computer, with no need for a printout. In that case, layout format is less important.

The output arrangement is controlled in three ways:

By specifying the appropriate output operation ("print", "dprt", or "iprt", among others).

By indicating the total number of positions the output should occupy on a page, and the number of significant digits or decimal places to be printed.

By using non-data output operations ("cr", "tab", "sc", "daprt") to return the carriage, tab to manually pre-set carriage positions, and print fixed headings.

Note: If both input and output are handled through the typewriter, they will appear together on the same paper. Therefore, allowance must be made for the input when planning the output layout.

NON-DATA OUTPUT

daprt

"daprt*", followed by a list of single letters, digits, or other legal symbols, each followed by (*), causes output of these characters. For example, the statement

```
daprt*n*e*x*t* *c*a*s*e**
```

causes

"next case"

to be output. Notice that the space between T and C is indicated by actually spacing once between the two (*) codes.

Whether any of the characters used are also names of variables or operations is irrelevant.

Neither the character (*) nor the formal controls such as carriage return, tabulate, upper case, lower case, and back space can be indicated directly. For these, special mnemonics are typed following the "daprt*": for example,

"cr*" represents carriage return, "uc*" and "lc*" represent upper case and lower case respectively. These mnemonics are listed in Table 2, Appendix B of this manual. Alternatively, any character can be output by giving its two-digit code shown in the same table. If the two-digit code does not correspond to a typewriter character, nothing will be printed if the typewriter is selected, but if the tape punch is selected, a code will be punched.

The two-digit and three-digit codes 64-127 do not represent characters, but input-output selection codes. Some of these codes cause selection of a specific input device and deselection of any previously selected input device. Others cause selection of a specific output device without causing deselection of other output devices previously selected. Some deselect all devices. Table 2 also provides the standard codes for the basic system (one typewriter, one tape-reader/punch).

reprt

n*reprt*, where N is in fixed-point form and "reprt*" is followed by a list of single letters, digits, or other legal symbols, each followed by (*), causes these characters to be output N times. The character representations are the same as for "daprt." For example, the statement

3*reprt* * *a* * * *b**

will cause this printout:

a b a b a b

Important: The "daprt" and "reprt" operations, each with its list of characters, must always appear as separate statements. No other operations may appear in the output statement.

cr tab sc

cr*, tab*, and sc* can be used as operations anywhere in the program, not just as codes after "daprt". They are used to cause carriage returns, tabulation on the typewriter, or printing of the stop code (*); or—if the output device selected at the time is the tape punch—to punch the corresponding codes on tape.

The other two-letter "daprt" format codes can not be used independently as operations.

aprt

If a group of 5 or fewer characters were input, earlier, by aread*b*, then aprt*b* would output the same characters. If fewer than 5 characters were read in and output is on the typewriter, only the actual number of characters read in will be typed. When output is on paper tape, 5 codes will be punched in every case, with initial tape feed characters (no holes punched) making up any blank character positions for a total of 5. (When such a tape is printed out, tape feeds are ignored.)

DATA OUTPUT

print

n*print*b*, where N is in fixed-point form and B is in floating-point form, causes the value of B to be output in floating-point notation. The value of the integer N controls the number of digits which will be printed, and the total space taken up on the page. Specifically, assume N is 100C + F, where C is the field width (i.e., the total number of characters to be output), and F is the

number of digits following the decimal point. Then the printout will appear in the form: (C - F - 6) leading spaces; the sign of the number (space if plus, (-) if minus); the decimal point; F digits after the point; the letter E; and sign and two digits of the exponent or power of 10 by which the fraction must be multiplied—a total of C characters. If any digits are to print, C must be greater than 6 and F should be less than (C - 6).

For example, if N = 1605 and B is the floating-point number 123.45678, then n*print*b* (or 1605*print*b*) will cause the output

.12346E+03

preceded by 6 spaces, including one for the sign. The printout is rounded to the number of digits specified: .12346, not .12345, since the next digit (6) is more than 4.

dprt

n*dprt*b*, where N is in fixed-point form and B is in floating-point form, causes the value of B to be output in ordinary decimal form. ("dprt" is read "decimal print".) Assume N is 100C + F, where C and F are integers; C is the field width, and F is the number of digits following the decimal point. The printout will appear in the form (C - F - 2) digit positions before the decimal point, with spaces for the leading zeros; the sign of the number (space for plus, (-) for minus) inserted before the first non-zero digit; decimal point; and F digits after the decimal point—a total of C characters.

If the number B is too large to be printed in the format indicated by N (that is, if it has more than F - C - 2 digits in front of the decimal point), the number of decimal places printed, F, will be reduced as much as necessary. If there is not enough space even with F = 0, the output field of C typing positions is filled with slashes (/).

If N = 1602 and B is the floating-point number -987.65432, then n*dprt*b* (or 1602*dprt*b*) will cause the output

-987.65

preceded by 9 spaces.

iprt

n*iprt*b*, where N and B are both in fixed-point form, causes the value of the integer B to be output. If N = 100C, where C is an integer indicating the field width, the value of B will print without a decimal point, preceded by leading spaces and then the sign (space for plus, (-) for minus), for a total of C characters. If N is 100C + F, where F is an integer between 1 and 8, a decimal point will be printed before the last F digits.

If the number B is too large to be printed in the format indicated by the value of N, the field width is increased to accommodate the number. If the two leading digits of the too-large number are 10, 11, 12, 13, 14, or 15, one added space is saved by printing the letter A, B, C, D, E, or F in place of the two leading digits.

check

An optional print operation is provided in ACT IV for testing programs and for monitoring the progress of long computations. If a statement such as

ss1*check*13**

is written after an assignment or other statement which has a "result," this check statement has no effect if SENSE SWITCH 1 is OFF during execution of the statement. But if the Switch is ON, the typewriter will print "@ 13" on a new line, followed by the value of the result of the preceding statement. If the preceding statement has a floating-point result, the check line number (here 13) must be positive. If the result is in fixed-point form, a negative check line number must be used.

Any Sense Switch (1, 2, 4, 8, 16, or 32) can be specified.

The check expression does not have to be a separate statement; it can be written as the last part of the statement to be checked.

hxpch

hxpch*b* produces the value of B in hexadecimal form, followed by (*). Here B can have any form internally: fixed-point, floating-point, or the form in which "aread" stores its input.

The hexadecimal form, which consists of 8 characters per word, chosen from the numerals 0 to 9 and the letters A to F, is not easily interpreted. The principal use of "hxpch" is to punch on tape intermediate results of a computation so that they may be input later for another computation, using "rdhex". The "hxpch-rdhex" input/output combination is faster than any other, is applicable to any type of variable, and introduces no conversion errors in transfer from internal to external form. When the "hxpch-rdhex" combination is to be used, the first program must contain a series of "hxpch" operations to punch out the values of the variables which are to be input later, and the second program must contain a series of "rdhex" operations in corresponding sequence to read them back into the computer.

FLOW CONTROL

LABELS

Statements are normally executed in the order in which they appear in a program. If the flow of control is to be modified, it is necessary to label some of the statements.

A statement label can be any name of up to 5 letters which has not already been reserved for special use in the ACT IV system. Table 3, Appendix B, gives the complete list of such restricted names. In addition, if a name is already used for a variable, it can not also be used as a statement label.

To label a statement, precede it by the name of the label, two periods, and (*). It is customary to type each statement on a separate line. If it is not preceded by a label, depressing the "tab" key on the typewriter will produce an indentation to indicate this. For labelled statements, it is common practice to type the label (including ".. *") at the margin, and then tab. Thus, all statements start at the same position on the page, and all labels are displayed in the left margin. These typing conventions merely assure legibility and uniformity. Computation is not effected by such forms of copy arrangement, since the "tab" code does not enter the computer on input.

THE PRIMARY TRANSFER OPERATIONS

When discussing statement labels in the following operations, the ". . " are not included. These periods are used only at the time when the statement is labelled.

use

If HERE is a statement label, use*here* causes the normal flow to be broken. The computer goes back or ahead to the statement so labelled, and sequential execution of statements resumes at the new point.

In contrast to other operations which change the flow, "use" is unconditional; that is, it is carried out regardless of any conditions which may exist.

if*a*neg*
if*a*zero*
if*a*pos*

If HERE is a statement label, and A is a variable or a computed quantity, then

if*a*neg*here*

causes the normal flow to be broken by a transfer to the statement HERE, if the value of A is negative. Similarly, if*a*zero*here* causes the flow to be broken by a transfer to HERE if the value of A is exactly zero; and if*a*pos*here* does likewise if A is positive (not zero or negative).

Subscripted switch vectors, which are discussed in greater detail in the next section, can be used in place of simple statement labels with NEG and POS, but not with ZERO.

If more than one test is to be made on the quantity A, they can be combined. Suppose that, if A is negative, the program flow is to jump to a statement labelled CASE1; if zero to CASE2; and if positive, to SPEC. The single statement

if*a*neg*case1*zero*case2*pos*spec**

will make the three-way test. The NEG and ZERO could be in either order, but POS must come last if it is included. However, this statement is slightly less efficient than

if*a*neg*case1*zero*case2*use*spec**,

where the last test is replaced by USE, resulting in a shorter and generally faster program. If SPEC is the statement immediately after this test statement, the "use*spec*" can be omitted entirely.

In place of A in these examples, any fixed-point or floating-point expression can appear. For example, to transfer to the statement HERE if the fixed-point variables I and N are equal,

if*i*i-*n*zero*here**

or, more readably,

if*[*i*i-*n*]*zero*here**

can be used.

Because of the round-off errors of floating-point arithmetic, zero-testing of floating-point quantities is not generally predictable. For example, if the value EX was obtained by adding the floating-point number 0. 1 ten times, the result will not exactly equal 1. 0. Therefore,

```
if*[*ex*-*1. 0*]*zero*here**
```

will not result in a transfer of flow to the statement HERE.

LOOPING

step until rpeat

The statement

```
i*step*j*until*n*rpeat*alpha**
```

has the following effect. First I (which represents any fixed-point variable) is incremented by the amount J, just as if

```
i*i+*j*=*i**
```

had been written. If this step carries the value of I past the value of the fixed-point quantity N, control flows to the next statement in sequence. Otherwise, control flows to the statement labelled ALPHA.

Here J represents any fixed-point expression or variable, which may be positive, negative or zero. (In most applications the integer 1 is used.) If J = 0, control always flows to the next statement in sequence. The N represents any fixed-point variable or expression.

The simplest application of "step" statements is in programming loops. See some of the sample programs in Part 3 of this manual.

"step" statements should not be used for stepping and testing floating-point variables.

It is permissible to begin step statements with the word "for", thus: for*i*step*, etc. This maintains consistency with ACT III. If "for" is used, it should not also be used as a region or procedure name in the program.

OTHER OPERATIONS

ss1*bcon*
ss2*bcon*
ss4*bcon*
ss8*bcon*
ss16*bcon*
ss32*bcon*

If HERE is a statement label, then ss1*bcon*here* causes a transfer to the statement so labelled, when SENSE SWITCH 1 on the RPC-4000 control panel is depressed. If the Sense Switch is up, the computer continues in normal sequence instead. The other five instructions test SENSE SWITCHES 2, 4, 8, 16, and 32 respectively. By setting the Sense Switches appropriately, the computer operator can choose among several options in a program which contains these Switch Test operations.

ss64*bcon*

ss64 is a non-readiness query for the High-Speed Reader. If HERE is a statement label, then ss64*bcon*here* causes a transfer to the statement so labelled when the photo-reader is the selected input device but is not ready for use.

read*a*bcon*
iread*a*bcon*

If HERE is a statement label, then bcon*here* appearing after a "read" or "iread" operation will cause a transfer to the statement HERE when the

expression read contains the letter F, as in f*, "final*", or "end of data*." As long as the expression read contains a legitimate number, the transfer operation has no effect. Thus, the end of data can be indicated easily on the data tape itself; the program can detect this end code and take appropriate action.

stop

The "stop" operation halts the computer after a program has been executed. It does not interrupt the translation of the program from ACT IV language to RPC-4000 machine language.

The last operation in an ACT IV program must normally be either a "use" operation, returning the flow of control to an earlier statement, or a "stop". Otherwise, the computer runs on, with unpredictable results, after execution of the last statement of a program.

The "stop" operation may also be used during execution of a program to allow the operator to insert another tape, reset Sense Switches, or the like. In the latter case, it is suggested that "daprt" be used in a statement before the stop, to print out a brief reminder to the operator about the purpose of the upcoming stop. After the computer has stopped, depressing the START switch on the RPC-4000 control panel allows the calculation to continue at the next operation.

go to

"go to" is an operation which indicates a variable transfer, or statement switch, which can be set to different values at different points in the program. "go to" appears only in the form of a labelled statement sw1. . *go to**. (The form go to*s0** may be used for compatibility with ACT III for the LGP-30, but "s0" is meaningless in ACT IV.)

Note the single space in "go to". It is part of the name of this operation, and can not be omitted.

to

If SW3 is the label of a switch statement "go to**", and HERE is the label of another statement, then

sw3*to*here*

sets the switch statement so that is has the effect (but not the internal form) of

use*here*

It is permissible to begin with "set*", for consistency with ACT III. Thus: set*sw3*to*here* If this is done, "set" can not be used as a region or procedure name in the program. If a switch "go to" is reached due to a programming error before it has been set, it has the effect of a dynamic stop; that is, the computer repeatedly and endlessly executes the same machine instruction, and the displays on the RPC-4000 control panel do not change.

Note: If HERE is a switch statement, then

sw3*to*here*

must be used with caution. If the program contains a sequence

```
                here*to*alpha**
                . . . . . .
                sw3*to*here**
                . . . . . .
                here*to*beta**
                . . . . . .
sw3. . *        go to**
```

then the effect of SW3 is use*alpha**, not use*beta**. (ACT IV differs from ACT III in this respect.)

REGIONS AND SUBSCRIPTS

Data is often handled in blocks, using a single name for the entire block and calling individual positions in the block by a serial number.

SIMPLE REGIONS

dim

The statement dim*a*55** informs the processor that A is to be the name of a block or region of at most 55 positions. Any number of such declarations can be incorporated into a single dimension statement: thus dim*a*55*list*37** indicates that A is the name of a block of at most 55 positions, and LIST of another with at most 37 positions. All dimension declarations should be given at the beginning of the program. Otherwise incorrect machine language programs may result, especially in long programs, since the processor does not check the availability of computer locations which are to form the region.

The number of positions to be reserved must be given as an explicit integer constant. A form such as dim*a*n**, where the dimension is given as a variable, is not allowed. When region space is allocated for use by a program, the largest size needed should be specified. However, such region size declarations should be governed by the space available in the RPC-4000.

SUBSCRIPTS

Region names without subscripts are of limited use. A region name followed by a subscript, on the other hand, can be used almost everywhere that a variable can. The only exceptions are that a subscript can not be merely the name of another subscripted region nor can it be a formal parameter of a procedure. Subscripts must be fixed-point numbers or expressions. If 55 locations have been reserved for the region A, they can be referred to by the subscripted names a*0*, a*1*, . . . , a*54*. If the programmer whishes to have a region TABLE and refer to the positions in this region as table*1* through table*10*, he will have to reserve 11 locations for this region,

dim*table*11**,

since the processor will reserve a place for table*0* whether or not the programmer intends to use it.

Any fixed-point variable or expression can be used as a subscript in place of constants, provided its value is in the range allowed for the particular region. While subscripted region names can not be used alone as subscripts, arithmetic expressions containing operations which use subscripted region names are allowed.

Thus if I and J are variables in fixed-point form and B is a region name, any of the following are permissible ways of referring to an individual position in the region A:

```
a*i*                    a*[*i*]*
a*[*i*i+*j*]*           a*[*B*i*ix*j*i-*7*]* .
```

Notice: if the subscript is more than a single constant or variable, it must be in brackets; if it is only a single variable, it may be in brackets.

MATRICES (TWO-SUBSCRIPT REGIONS)

dim
,

A statement of the form dim*box*7*, *10** informs the processor that the name BOX is to be the name of a region of 70 (=7x10) locations to be referred to as

```
box*[*1*, *1*]*    box*[*1*, *2*]*   . . .   box*[*1*, *10*]*
box*[*2*, *1*]*    box*[*2*, *2*]*   . . .   box*[*2*, *10*]*
. . . . . .
box*[*7*, *1*]*    box*[*7*, *2*]*   . . .   box*[*7*, *10*]*
```

In place of integer constants in the brackets, any fixed-point variables or expressions can be used, provided their values are in the range allowed for the corresponding subscript for the particular matrix.

A single dimension statement can contain a mixture of declarations of simple regions and two-subscript regions:

```
dim*a*55*list*37*box*7*, *10*ww*10**
```

dimensions three simple regions (A, LIST, WW) and one matrix (BOX). The occurrence of a comma and second integer distinguishes the matrix names.

SWITCH VECTORS

dfine

The "dfine" code makes it possible to control the relative positions in the RPC-4000 memory to which the processor assigns variables or statements. A complete description of the meaning of this code will be given in Part 2 of this manual. Let it be merely indicated, not explained, at this point, how it can be used to construct a switch vector.

Suppose ALPHA, BETA, and GAMMA are the labels of three statements, perhaps the beginnings of three alternative branches of the program. If the programmer wants to be able to calculate which branch to take by referring to the three alternatives as "swvec*1*", "swvec*2*", and "swvec*3*" respectively, he could use

```
use*swvec*n**
```

to transfer to one of the branches according to the value of the fixed-point variable N.

To set up ALPHA, BETA, GAMMA as a switch vector "swvec", the following two statements should be written with the "dim" statements at the beginning of the program:

```
dfine*gamma*1*beta*1*alpha*1**
dim*swvec*1**
```

Note that the switch vector is written in reverse order (i.e., GAMMA is switch vector 3 and ALPHA is switch vector 1), and that "1" appears after each name including "swvec".

MISCELLANEOUS OPERATIONS

wait

The command wait, followed by a single (*), informs the processor that the end of an ACT IV-language program has been reached, and no further translations are to be done. Other uses of "wait" and another way of terminating ACT IV programs—namely, with "xeq"—are explained in Part 2 of this manual.

comnt

Explanatory and identifying comments can be included anywhere in an ACT IV program or before the first statement. A comment statement may consist of any characters excepting (*). It must be preceded by the code "comnt*", and followed by two (*) codes. For example:

```
comnt*          The program will now call for the input of
                n+1 pairs of numbers (x, y).  **
```

It is recommended that every program begin with a comment statement giving the name of the program, the author's name, and the date the program was written or last revised.

"comnt*", followed by a comment and a single (*), can also be used at any point inside any statement, except after "daprt". "comnt*" instructs the processor to ignore what follows up to the next (*). When the program tape is printed out during its conversion to machine language or off-line, the comments can be read by the programmer or operator, but have no other effect.

prev

"prev*" yields the result of the last calculation, so that it can be used as if it had been given a name. To understand the meaning of "last calculation" the scanning method of the processor must be kept in mind, and how its precedence rules are implemented. This is illustrated by the following examples:

```
prev*=*a**              assigns to the variable A the result of the
                        preceding statement executed, if it had a
                        result.

sqrt*[*prev*]*=*b**     assigns to the variable B the square root of the
                        result of the preceding statement executed, if
                        it had a result in floating-point form. Because
                        "sqrt" and "prev" have the same precedence,
                        there would be a conflict, unless "prev*"
                        was enclosed in brackets.

[* ... ]*x*prev*=*c***  assigns to C the square of the floating-point
                        quantity calculated by the expression in the
                        brackets.
```

MULTI-PURPOSE STATEMENTS

Statements in an ACT IV program were primarily intended to be units with a single effect. Thus the statement

```
a*+*b*=*r**
```

has the single effect of assigning a value to the variable R. The statement

```
1606*print*[*a*+*b*]**
```

has the single effect of printing the value of a certain expression. The statement

```
if*[*a*+*b*]*neg*here*use*there**
```

has the single effect of "branching" according to the sign of the value of a certain expression. The statement

```
cr**
```

has the single effect of returning the typewriter carriage.

However, it is possible to write statements which have several effects. Multi-purpose statements are useful for grouping closely related functions together on one line. In some situations the saving of a few typed characters is possible, and translating time is reduced when fewer statement addresses have to be printed out by the processor.

On the other hand, some disadvantages must be weighted when considering the use of multi-purpose statements. They may result in a longer machine-language program than if the statements had been split. More important, the strict application of the precedence rules by the processor may result in unexpected effects and incorrect results.

The examples below demonstrate the proper and improper use of multi-purpose statements.

Example 1 Several input operations, as well as "cr" and "tab", can be combined in one statement.

```
cr*iread*n*tab*read*a*tab*read*b*cr**
```

is acceptable.

Example 2 Several output operations can be combined, as well as "cr" and "tab" (but not "daprt", which must always form a statement by itself).

```
cr*0400*iprt*n*1203*dprt*[*a*+*b*]*1203*dprt*[*a*-*b*]*cr**
```

is acceptable.

Example 3 When brackets are present, input and output should not be mixed. The statement

```
read*a*read*b*1606*print*[*a*+*b*]**
```

will not work as probably intended, since the sum A + B will be cal-

culated on the basis of whatever values A and B had before the new A and B values are read in.

Example 4 Input operations such as "read*a*", "iread*a*" not only cause the value input to be assigned to the variable A, but also have that value as a result which can be used in an expression. Thus the two statements

```
read*a**
b*+*a*=*s**
```

can be combined into

```
b*+*[*read*a*]*=*s**
```

The latter statement has a double effect: it changes the value of A by input, and the value of S by evaluation.

A form such as b*+*read*=*s**, in which "read" is not followed by a variable, is not correct.

Care should be used if the variable read in occurs more than once in the expression. For example, in

```
[*read*a*]*x*[*a*+*b*]*=*s**
```

the [a+b] will be calculated before input takes place, since input (and output) follow addition and subtraction in order of precedence of operations.

Example 5 Multiple assignments of the form

```
a*+*. 001*=*a*=*red**
```

have already been mentioned: the above would assign the former value, A plus 001, to both A and RED. A number of separate assignments can also be combined in one statement.

```
0*=*a*=*b*1*=*i*=*j*1. *=*s**
```

assigns zero to A and B, fixed-point 1 to I and J, and floating-point 1 to S. (If B is a region name, the above analysis is wrong: zero would be assigned to A, B_1, I, and J.) Combinations like this are often used to put all initializations into a single statement for brevity.

Example 6 The value of an expression can be assigned and used simultaneously, as in the examples

```
c*x*[*a*+*b*=*s*]*=*p**
```

and

```
1606*print*[*a*+*b*=*s*]**
```

There is an exception: if the expression assigned is merely a constant or variable in floating-point form, the result of the assignment can not be used for floating-point operations in that statement. Thus

```
[*3. 1416*=*pi*]*x*d*=*circm**
```

is not correct, nor would it be with PIE in place of "3. 1416" in the statement.

Example 7 The statement

```
1606*print*[*a*+*b*]*=*s**
```

is not correct. Technically, (=) yields a result, the value assigned, which can be used further; but PRINT has no usable result within the computer.

PROCEDURES

THE SUBROUTINE CONCEPT

A subroutine or subprogram consists of a group of statements which is either to be used several times within the same program, or appears sufficiently useful to be preserved for incorporation into future programs.

A subroutine can be written as an integral part of a program. It can be used at several different points of the program if a switch "go to" is the last statement and is set each time before the program transfers to the start of the subroutine. When a subroutine is used, the following must be carefully observed:

Names and labels used in the program may not be applied to different items in the subroutine.

All quantities dealt with by program and subroutine must be referred to by the same names. Thus, if a matrix inversion subroutine is written to invert the matrix MAT1 and place the inverse in MAT2, any matrix to be inverted would first have to be copied by a programmed loop into the MAT1 region, and the resulting inverse would then have to be similarly removed from MAT2 to the region in which it is wanted.

THE PROCEDURE CONCEPT

The term "procedure" in ACT IV refers to a more formal type of subroutine which is not subject to the same restrictions as ordinary subroutines.

No conflict is possible between statement labels and variable names introduced within a procedure and the same labels and names used later, because the ACT IV processor "forgets" their definitions after it finishes translating the procedure.

At each point in the main program where a procedure is to be used, it can be "told" which variables and regions it is to work with, and where to put the results.

Procedures can be called on from within other procedures, and this process can be stacked to any depth. However, recursive procedures—procedures which call on themselves, directly or indirectly—are not allowed in ACT IV.

USING A PROCEDURE

A procedure is used to work on some data and produce some answer or answers. Each item of data will be a single quantity—such as can be named by a variable—, or a region which is identified by its name, with no subscripts given.

Suppose there is a procedure for moving from one region into another those numbers whose value exceeds a limit. The procedure must be told how many numbers there are to be examined, in what region they lie, what the limit value is, and what region is to receive the removed numbers. In addition, it would be useful to know how many numbers are left in the original list.

The description of the procedure will contain the name of the procedure, which is used when calling on it, and, in a specific order, the types of names or quantities which must be specified to give the procedure the information it needs.

The list-splitting procedure might have the following description, in part:

name:	split
supply:	1) length of original list, in fixed-point form; 2) name of original region; 3) value of limit, in floating-point form; 4) name of region to receive the removed numbers.
result:	length of reduced list, in fixed-point form.

Supposing a main program states this problem as: Take the numbers greater than 1000. 0 out of the list of 2k-1 numbers which are in the first 2k-1 places of the region ALL, put them into region BIG, and set LEFT equal to the number of values remaining in ALL afterwards.

Then, only the procedure statement:

```
split*[*2*ix*k*i-*1*]*, *all*, *1000. 0*, *big*=*left**
```

needs to be written. A dimension statement at the beginning of the main program must have reserved enough locations for the longest possible list to go into BIG.

Note: the procedure statement consists of the name of the procedure being called on, followed by the list of actual "parameters" to be supplied, in the order specified by the description of the procedure in question. The first parameter in the example, the length, is the fixed-point expression [2k-1] (K is assumed to be in fixed-point form.) If the length of the list were given as a constant (37) or a variable (N), the brackets would not be necessary. The second parameter is the name of the first region. The third parameter is the limit, in this case a constant.

So far, these are input parameters: they give information to the procedure. Input parameters can be constants, variables (including subscripted region names), expressions, region names, statement labels, or switch vector names with or without subscripts.

The fourth parameter in the example is BIG, the name of a region which the procedure will change: hence it is an output parameter. Only region names are allowed as output parameters, or as parameters which play both input and output roles. (The second parameter in the list, the name of the orginal region, is an output parameter as well as an input parameter, since the original region is changed by the procedure.)

In addition, a procedure can have a "result": a single value, in fixed- or floating-point form as shown in the procedure description. If a procedure has

a result, the result can be used only as shown in the example, by assigning it immediately:

```
. . .        =*left**
```

It can not be used as a term in a calculation: a statement like

```
split*[*2*ix*k*i-*1*]*, *all*, *1000. 0*, *big*, *-*list*=*new**
```

will not operate correctly.

Of course no (=) operation would be written when calling into use a procedure which has no "result" shown in its description.

Note that a procedure could report on an exceptional situation by executing a transfer—out of the usual sequence of control—to a statement whose label is an input parameter. This is why input parameters which are labels can occur in some procedures.

,

In the example, the parameters in the list were separated by commas. The use of commas is optional, except that they are always required to separate a parameter which is a region (or switch vector) name from the following parameter. The first and third commas could have been omitted, but not the second. (In the example, all three were retained for clarity and uniformity.)

One more example: If a "split" procedure is to remove from the first 37 numbers in the region W those which are larger than the number in the m'th location of region Y, and put them in region Z, the procedure statement could be written:

```
split*37*w*, *y*m*z*, *=*left**
```

Notice that Y_m is simply specified as the limit value in the form y*m*. There is no comma after the Y here, since Y is not the parameter, but y*m*. The comma after W is necessary, since this region name is a parameter.

CONSTRUCTION OF PROCEDURES

This section will show how to write a procedure.

enter

When using a procedure, the list of actual parameters is written after the name of the procedure. The first line of the procedure itself, called the "enter" line, is rather similar. It consists of the code "enter*", followed by the name of the procedure and by the names of the parameters as they will be used in the procedure definition. The line ends with an extra (*) code after the last parameter name, giving it the form of a statement.

The choice of procedure and parameter names is subject to the same restrictions as apply to names of variables. In particular, the name of a procedure can not have more than 5 characters.

Parameter names used within the body of a procedure are called the formal parameters. When the procedure is called into use, the formal parameters are, in effect, replaced by the actual parameters specified in the procedure statement.

The formal parameters must be simple variables: subscripted region names are not allowed.

array

If any of the formal parameters are to represent names of regions, the "enter" declaration must be followed by another line. This line begins with the code "array*", followed by the names of those formal parameters which represent names of regions.

The "enter" line (and "array" line, if one is required) constitute the procedure heading. This is followed by the actual procedure in the form of ACT IV statements operating on the formal parameters, and using any convenient names for intermediate variables or regions which may be needed. As was stated earlier, there will be no confusion between the variables used inside the procedure and those introduced later, so no effort need be made to avoid duplication of names.

No "dim" statements should be given inside the procedure for the "array" parameters: space reservations for these is provided by the "dim" statements given for the actual region parameters in the main program.

exit

In most cases the procedure is intended to complete its function and then allow the flow of control to return to the main program, to the point following the call for the procedure. This return transfer is provided by the statement

```
exit**
```

used at the point in the procedure where the return to the main program is to take place. This is often the last statement of the procedure. However, a procedure may also have the form of a loop, with "exit**" appearing in the midst of the physical list of statements, constituting one outcome of a test.

If the procedure has a number of branches, several "exit**" statements may be present, one at the end of each branch.

The result of the procedure, if one is to be announced, is the result of the last statement executed before the "exit**". If the desired result can be conveniently calculated as the last step before exiting, there is no particular difficulty. If the result quantity had to be calculated at an earlier point in the procedure, and therefore given a name such as XYZ, place the statement

```
xyz*=*xyz**
```

just before the exit, since this assignment statement does have XYZ as result.

end

The last statement of the procedure (last in order of typing, not in order of execution) must be followed by the code "end**". This notifies the processor that the list of statements constituting the procedure have all been input for translation, and that the terminal processing—primarily, the "forgetting" of all names of variables and statement labels introduced in the procedure—should take place.

Many procedures end with the two lines

```
exit**
end**
```

The "exit**" indicates the dynamic end of the procedure, the point at which its

execution is to end; "end**" indicates the physical end of the procedure, where translation into RPC-4000 machine language is to end.

PLACING OF PROCEDURES

The following is a recommended plan for placing procedures in programs. If the main program does not use "dim" or "dfine", all procedures used by the program may be written first, then the main program. (If any procedures call upon others, the ones called upon must come first.) If procedures to be used are on separate tapes, they can either be copied into a single tape, off-line, then followed by the main program, or each tape can be translated in turn. This is further discussed in Part 2 of this manual.

If "dim" or "dfine" is used, the total number of locations to be reserved must be established first, including the 1's in the "dfine" statements. Supposing the total is 874 the program should begin with the statement

```
dim*dummy*874**
```

dummy

A convention exists that the name "dummy" is not to be used for any other purpose in any program or procedure; but any name not otherwise used may be written in its place. Thus, "dummy" or its substitute is written and followed by the procedures used. The last procedure is followed by the command

```
reglo*04200**
```

(The special meaning of the number 04200 will be explained further in this subsection and in Part 2. 04200 is the number to be used normally.)

Next, the main program is written, starting as usual with all the "dim" and "dfine" statements required.

The following facts should be considered when working out a modification of the suggested plan, above, for using procedures in a main program.

(1) At the "end**" of each procedure, the processor "forgets" the names of all variables, etc., introduced within the procedure, but not those introduced before (if any). If names are introduced before the "enter" of a procedure, and the same name happens to be used inside the procedure (a question often inconvenient to check), the double use of the name is almost certain to lead to error.

(2) Space reserved by "dim" and "dfine" is assigned backward from location 04200 in the normal mode of program translation. No test is made whether any of the space required for a region is already used for other purposes.

Because of (1), the dimension statements of the main program should not be written before the procedures. Because of (2), the procedures should not be written before the dimension statements, unless some way is provided to prevent overlap.

The dim*dummy*874** reserves all the region-space the main program will need, from 04200 backward. If the procedures themselves contain "dim" or "dfine" (unusual, but possible), the space assigned for these will automatically fall before the space for the main program's regions.

The reglo*04200* command resets the region location register within the

processor to the value it had at the beginning. The following "dim" and "dfine" statements in the main program will cause the actual regions of the main program to be located within the space reserved at the beginning as the dummy region "dummy".

Instead of the method of avoiding conflicts described above, the commands "res" and "avl" defined in Part 2 could be used to reserve or make available different areas of memory for the procedures and for the main program and its regions.

OPERATING INSTRUCTIONS 2

GENERAL

This part explains the operation of the RPC-4000 for translating and running ACT IV programs. Since some of the commands which control translation and computer operation may be incorporated into the ACT IV program proper, it is suggested that this part be scanned even by programmers who do not intend to operate the computer themselves.

A hexadecimal tape of the processor is available which can be loaded into the RPC-4000 by the usual bootstrap procedure. This takes about 15 minutes with the standard 60-character/second tape reader.

The program to be translated can be on tape, or it may be typed in. In the normal "load-and-go" mode of operation, the translation will be stored in the computer and not punched out.

Any translated part of a program can be executed immediately. Errors discovered at this time can be corrected, and translation can be continued. In the load-and-go mode, even the completed program can call for the input of statements in ACT IV language which can be executed subsequently in the program. Thus, a data tape may consist of numbers intermixed with ACT IV statements.

If a program is to be re-used in the future, a tape of its translated form can be punched out during the translation process. Corrections made at this time, including those discovered while executing portions of the program, will be recorded on the tape. The resultant tape can be read in at a later time for executing the program in its corrected form. Further corrections are not possible, however. Calls for input of ACT IV statements are not allowed within the program if it is to be run, later, without retranslation.

If the 42 tracks available for a program in load-and-go mode are not enough, 53 more tracks can be made available by switching to the "punch/do not store" mode. When the translation is finished, the punched part of the program can be read in to overlie the translating portion of the processor. The entire program can be executed, but correcting in ACT IV language obviously becomes impossible, since the corrections could not be translated.

STORING THE PROCESSOR

The following operating instructions are given here as a practical extension of the programming procedures discussed in Part 1. To keep this section as simple as possible, it is keyed to the basic RPC-4000 system using the typewriter and tape reader/punch. As a further simplification, the following abbreviations will be used throughout the subsequent discussion:

CC:	computer control panel
RPR:	reader/punch cabinet, right-hand panel
RPUL:	reader/punch cabinet, upper half of left-hand panel
RPLL:	reader/punch cabinet, lower half of left-hand panel

If the light underneath a control switch is in the wrong state according to proper procedure—OFF instead of ON, or vice versa—this can be corrected in most cases by depressing the switch. It should not be touched when it is in the correct state. Exceptions to this rule will be noted as they occur.

The following steps will activate the computer and lead to storage of the ACT IV processor:

(1)	CC	POWER ON: ON Both memory protection switches (behind the slide below the oscilloscope display) UP.
(2)	RPUL	Make sure the POWER ON switch is UP.
(3)	RPR	SYSTEM POWER: ON
(4)	RPR	SINGLE CHARACTER MODE: OFF PARITY MONITOR RESET: OFF PARITY MONITOR INHIBIT: OFF
(5)	RPR	Depress MASTER RESET
(6)	RPUL	POWER ON: ON
(7)	RPLL	TYPEWRITER SELECT: OFF READER SELECT: OFF PUNCH SELECT: OFF
(8)	RP	Place the processor (hexadecimal tape) in the reader.
(9)	CC	If the computer was just turned on, the console may not yet be lit. Wait until the STOP light is ON.
(10)	RPUL	READER TO COMPUTER: ON
(11)	CC	ONE OPERATION: ON Depress SET INPUT MODE EXECUTE LOWER: ON Depress START COMPUTE (The tape will move a short distance—one word—and stop.)
(12)	CC	Depress START COMPUTE again Depress EXECUTE LOWER (to turn it OFF) Depress SET INPUT MODE again Depress ONE OPERATION (to turn it OFF) Depress START COMPUTE

The tape will start moving again and continue for about 15 minutes, storing the entire processor in the computer. When input is finished, the tape and computer will both stop.

STARTING TRANSLATION

In order to start translation, the following steps are required:

(1) Store the processor. It is not necessary to repeat this if another

program has just been translated and/or run, unless that program had been too long and so was translated in the mode which overlays part of the processor.

(2) RPR — Depress MASTER RESET

(3) If the ACT IV language program is on tape, place the tape in the reader. If the program—or at least its beginning—is to be entered from the typewriter, this input device will have to be selected manually at the appropriate time (see step 12).

(4) Set two tabs on the typewriter: one about 10 places from the left margin, and one the same distance from the right margin.

(5) RPUL — TYPEWRITER TO COMPUTER: ON

(6) CC — Normally, all SENSE SWITCHES: OFF.
But:
If the entire translated program is to be punched out,
SENSE SWITCH 32 ON;

If none of the translated program is to be stored in the computer while translating and punching,
SENSE SWITCH 16 ON
(no effect unless SS 32 is ON also);

If no printouts of statement locations are wanted during translation,
SENSE SWITCH 8 ON.

(Switches 4, 2, 1 are not examined while translating.)

(7) CC — ONE OPERATION: ON
Depress SET INPUT MODE
EXECUTE LOWER: ON
Depress START COMPUTE

(8) The light on the typewriter will come on. Return the carriage and type
ACT4*
The light on the typewriter will go out.

(9) CC — EXECUTE LOWER: OFF
ONE OPERATION: OFF
Depress START COMPUTE

(10) The following will happen almost immediately:

If SENSE SWITCH 32 is ON, a leader and then the code FILL* will be punched on tape.

The reader will be selected for input.

The typewriter will be selected for output.

Input Duplication Mode will be selected.

The computer will run for about 15 seconds, during which the processor initializes the various tables it uses while translating. At the end of that time the typewriter tabs, a printout "0000" appears, and translation can begin.

(11) If the program is on tape and was placed in the tape reader in Step 3, it will start moving. Since Input Duplication Mode is selected by the processor, a copy of the program will print out simultaneously on the typewriter.

(12) If the program, or at least its beginning, is to be typed by hand, perform the following operations to change the selection of input device from reader to typewriter:

RPR	Depress MASTER RESET
RPUL	TYPEWRITER TO COMPUTER: ON
	COMPUTER TO TYPEWRITER: ON
RPR	Depress START READ.

The typewriter light will come on, indicating that the compiler is ready to accept the first word of the program. (A "word" is the sequence of characters up to and including an (*).) The light will go out when the (*) is typed. Do not resume typing until it comes on again a fraction of a second or so later. Continue typing the first statement, word by word.

After the input via the typewriter is completed, perform the following operations to allow input from the tape reader:

RPR	Depress MASTER RESET
	Position the tape in the reader.
RPUL	READER TO COMPUTER: ON
	COMPUTER TO TYPEWRITER: ON
RPR	Depress START READ.

(13) At the end of the statement there will be a longer pause than usual. If SENSE SWITCH 32 is ON, the translated form of the statement will punch out. Unless SENSE SWITCH 8 is ON, the typewriter will tab and print a 5-digit number which is the address, in decimal track-and-sector form, at which the translated form of that statement begins in the RPC-4000.

(14) The above is repeated, statement after statement, until the cycle is stopped by one of the processing commands "wait" or "xeq", or until the processor detects an error in the program.

NORMAL MODE PROCESSOR COMMANDS

Processor commands are distinguished from operations such as (+), "read", "use" and so on, because they control translation instead of becoming part of the translated program. (The line of separation is not always clear.) In this section, two commands will be defined which are used in normal load-and-go processing, as well as "loc" whose commonest use is in connection with one of these two commands. Other processing commands, usually used for correcting programs and for too-long programs, will be given later.

wait

When the processor encounters the line

wait*

instead of a statement it stops, or pauses, in the translation process. The computer STOP light comes on. Depressing START COMPUTE (RPR or CC) allows translation to resume, if this is wanted.

Notice that there is only a single (*) on a "wait" line.

Following are a few applications of "wait":

Either "wait" or "xeq" (described next in this section) should be at the end of the entire program, whether typed by hand or on tape.

If the program is on several pieces of tape, "wait" is punched at the end of each tape to give the operator an opportunity to remove it and insert the next part. In particular, tapes of procedures should always end with "wait". Long programs are preferably punched in several parts, so that it will not be necessary to copy the entire program if an error is noticed which the programmer wishes to correct off-line before translation.

If part of the program is on tape and part is to be typed by hand, "wait" should appear on the tape wherever a "switch to typed input" is desired. The pause would be used by the operator to change the input selection by these manual operations:

RPR		Depress MASTER RESET
RPUL		COMPUTER TO TYPEWRITER: ON
		TYPEWRITER TO COMPUTER: ON
RPR	(or CC)	Depress START COMPUTE to resume translation.

It is not necessary to use "wait" to go from typewriter input to tape input, since the processor already waits for typewriter input. When a "switch to tape input" is desired, do not type when the typewriter light comes on again, but:

RPR	Depress MASTER RESET
RPUL	COMPUTER TO TYPEWRITER: ON
	READER TO COMPUTER: ON
RPR	Depress START READ (not START COMPUTE)

xeq

If HERE is a statement label, the command

xeq*here**

causes translation to stop, and execution of the translated program to begin at the statement HERE, provided SENSE SWITCH 16 is OFF, indicating that the program is being stored in the computer. If SENSE SWITCH 32 is ON, meaning that the translated program is being punched out, the resulting tape will have a special transfer code at this point. When the tape is read in later, this code will interrupt the input process and cause a transfer to the translated statement HERE.

The statement before the "xeq" line must be a "use", "stop", or "go to". Otherwise the computer's execution of the program would continue beyond the end of the program with unpredictable results.

If the translated program is being punched out and "xeq" is used for testing purposes only, but is not wanted in the completed program, set SENSE SWITCH 32 OFF before typing "xeq". It can be turned back ON as soon as the transfer to the named statement has taken place.

loc

"loc*", followed by a computer address in decimal track-and-sector notation and by (*), can be used any place where a variable or a statement label might appear. One application is given below. Another use of "loc" might be for fitting instructions written in machine language into ACT IV-language programs.

When the ACT IV-language program is being translated, the computer address of the beginning of each statement is printed out unless SENSE SWITCH 8 had been depressed to suppress the printout. It has been shown how "xeq" is used to begin execution of the program at a labelled statement. Execution can also begin at unlabelled statements. For example, to start execution with the statement which is shown by the printout to begin at location 01234, input the command

xeq*loc*01234**

To start at the first statement of the program, normally 00000, input

xeq*loc*0**

DETECTED PROGRAM ERRORS

The processor can detect certain programming errors while translating the program. When it does so, it types out an identification of the error, and the computer stops. Typographical errors in the present statement are easiest to correct. Select typewriter input and depress START COMPUTE. The typewriter light will come on, and the processor will allow the retyping of the present statement correctly. More information on error correction appears in the next section. At this point, a list of possible error printouts from the processor are merely listed, and their meaning explained.

STATEMENT TOO LARGE COR. AND RESTART	A statement may have at most 63 (*) codes, not counting those following the brackets. If there are more, the statement must be split into parts. ("daprt" is not subject to this limitation: any number of characters can be listed in a single "daprt" statement.)
ILLEGAL SYM. COR. AND RESTART	The symbol just read is not a number, yet has more than 5 characters; or an invalid "daprt" code has been used.
TOO MANY [COR. AND RESTART or TOO MANY] COR. AND RESTART	The left and right brackets do not match properly.

LF. OP. INCORRECT OP. CODE IS ITEM NO. XX COR. AND RESTART or RT. OP. INCORRECT OP. CODE IS ITEM NO. XX COR. AND RESTART	One of the operations in the statement has no operand (or an improperly constructed operand) on its left or right. The operation in question is the XX'th word in the statement, not counting brackets or the statement label (if any).
DRUM IS FULL COR. AND RESTART	The space available to the program (initially, tracks 000-041) is used up. It is necessary to switch to the "punch/do not store" mode, and make more space available.
SYM. NOT DEFINED IN ENTER CORRECT PROCEDURE COR. AND RESTART	This printout is only obtained during an "array" declaration within a procedure. A symbol has been declared an array which was not listed in the "enter" line. If it is the array line which is in error, simply depress START COMPUTE and retype it. If the "enter" line was wrong, it is necessary to use "start" to back up the processor to the "enter" line.
SYMBOL TABLE IS FULL PUNT	No immediate correction is possible. Reduce the number of distinct symbols introduced by placing some of the variables in regions and by leaving out unnecessary statement labels, and start over from the beginning, "ACT4*".

ERROR CORRECTION

There are several situations in which error correction is possible.

When the program is typed in and the operator notices he made a mistake.

When the processor detects an error in a program being input from typewriter or tape.

When the program (or a part of it) is being run in "load-and-go" mode, and incorrect results point out an error.

By its very nature, error correction is almost always done by typewriter. Therefore, before attempting any of the correction methods below, the typewriter must be selected for input.

If the need for correction is discovered during the translation process, there is no difficulty in reaching the correction phase of the processor, known as ACTC. If the program is being executed in "load-and-go" mode, it is necessary to transfer to ACTC by manual operations:

CC	ONE OPERATION: ON Depress SET INPUT MODE EXECUTE LOWER: ON Depress START COMPUTE

The light on the typewriter will come on. Return the carriage and type ACTC*. The light on the typewriter will go out.

CC	EXECUTE LOWER: OFF ONE OPERATION: OFF Depress START COMPUTE

The computer will then transfer to the point in the processor which accepts new statements or corrections.

The error-correction methods below are applicable at various other times of program processing.

To correct any mistyped symbol if the error is noticed before typing the (*): Depress SET INPUT MODE on the computer control panel. This clears the Lower Accumulator. Look at the top line of the oscilloscope display (the Upper Accumulator). If any pattern is visible there, hold down SET INPUT MODE key and space the typewriter until the pattern moves completely out to the left. Then type the intended symbol correctly. (Caution: ONE OPERATION must be OFF when using this method—otherwise the SET INPUT MODE switch has another effect as well, which is not desired here.)

To correct the statement being typed, type at least 6 characters and a (*):

XXXXXXXX*

The computer will print ILLEGAL SYM. Depress START COMPUTE, ignore the address which will print out, and retype the entire statement correctly.

To correct the most recent statement after the second (*) is typed, or to correct the last few statements:

(1) If you have started typing another statement, cancel it by typing XXXXXX* as described above.

(2) See what address was printed by the processor after the first wrong statement. (This is the computer location where the translation of the statement begins.) Suppose it was 01111.

(3) Type start*01111**

(4) Retype the erroneous statement(s) correctly and continue.

An explanation of the effect of "start" is given at the end of this section.

To correct an earlier statement or statements:

(1) Finish the statement being typed.

(2) If the next statement is labelled, type the label with its two dots, and two (*) codes. If it is unlabelled, type one (*). A computer address will print out, say, 03333. This is where the next statement is to begin—note it carefully.

(3) See what address was printed by the processor after the first of the statements to be corrected. Suppose it was 01111.

(4) Type start*01111**

(5) Type the corrected statement or statements.

(6) See what address was printed out by the processor after the first correct statement following the one(s) just corrected. Suppose it was 02222.

(7) Type the statement use*loc*02222**

(8) Type start*03333** (the address noted in step 2).

(9) Continue typing the program where you stopped. If the next statement is labelled, the label does not have to be but can be repeated.

To insert a missing statement, check the two statements between which the insertion is to be made, to see which is easiest to retype. Then "correct" that statement by the technique given above, changing it into the missing statement(s) followed or preceded by a duplicate of the one being "corrected". Retyping one of the surrounding statements is unavoidable, in general, except by making a correction afterwards in machine language. This method falls beyond the scope of this manual and is therefore not explained.

start

The effect of start*01111** is to instruct the processor to place the first instruction of the translated form of the next statement into computer location 01111. It will do so even if 01111 has already been used. The subsequent instructions of the next statement will be in previously unused locations. If 01111 contained the first instruction of a former translated statement, all except that first instruction would remain in the computer unchanged but inaccessible.

CALLING THE TRANSLATOR BACK

Three operations are defined in this section which enable the translated program to call the translator back into use during the "load-and-go" mode of operation.

actc

When the statement use*actc** is executed in the program (not when it is being translated) the same point in the processor is reached as when transferring manually to make corrections (by typing actc* in conjunction with some button-pushing). If actc* is written before an "xeq" statement intended to test part of a program, it causes the computer to automatically exit and continue translation after it has run through the part of the program already input. To remove the use*actc** from the program automatically after making the test, label it, and

give the first statement after the "xeq" line the same label. The first instruction of the latter will replace the first instruction of the former in memory, and will effectively eliminate it.

actx
acte

During the execution of a program which was processed in the "load-and-go" mode, a statement in ACT IV language may be inserted to be executed immediately. For example, if HERE is a statement label, the statement

actx*to*here*use*acte**

calls for the input of one statement in ACT IV language (perhaps a heading or a sense switch option which vary depending on the data used with the program). That statement would be translated and executed immediately. Then the program would continue with the statement labelled HERE.

act4

If the statement use*act4** is executed in the program, the same point in the processor is reached as when transferring manually to ACT4*. The tables in the processor are initialized, and an entirely new program can be read in and translated. One use of this is to allow a semi-unattended translate-and-run operation of the computer: when each program has finished, it will automatically start the translation of the next program, if the programs and their data alternate properly on a single tape.

TOO-LENGTHY PROGRAMS

When the processor prints out DRUM IS FULL during the normal "load-and-go" translation mode, the following steps will more than double the available space, although the "load-and-go" feature must be sacrificed, including the ability to make corrections after trying part of the program.

(1) CC — SENSE SWITCH 32: ON (=Punch)
SENSE SWITCH 16: ON (=Donot store)

(2) If the translated program has not been punched out until now, prepare the tape:

RPLL — SELECT PUNCH: ON
SELECT TYPEWRITER: ON
Hold TAPE FEED until a foot or more of blank tape has been fed out as a leader.
Type FILL*
SELECT PUNCH: OFF
SELECT TYPEWRITER: OFF

(3) Make sure typewriter input to the computer is selected.

(4) RPR (or CC) — Depress START COMPUTE. Ignore the address which will print out.

(5) When the light on the typewriter comes on, type

avl*04200*09463**

(6) Repeat the last program statement, and continue to the end, returning to reader input when and if appropriate.

(7) When translation is finished, remove the newly-punched tape, and read it in.

The punched part of the program will overlay parts of the translating portion of the processor, which makes corrections impossible, once the tape is read in. If the program contains any use*actc** or similar statements calling on the translator, it can not be run at all.

Two alternate considerations should be introduced at this point:

If a program was extensively corrected before the DRUM IS FULL message, and if the remaining program appears to be no longer than what has been replaced by corrections, an alternative approach is available which preserves the convenience and correctability of "load-and-go". This method involves correction of the ACT IV-language program tape off-line and a new start. This utilizes the space in memory of those program parts which were later replaced by corrections; possibly accounting for the DRUM IS FULL message. Additional saving of space may be achieved in some cases by reducing the dimensions reserved for the regions.

If a program appears too long to fit in the normally available space—for example, if the regions alone take up the 42 x 64 = 2688 locations normally available—it is advisable to use the "punch/do not store" mode for the entire program to begin with. Having the extended space available enables the processor to optimize better, and results in a faster-running program. It is suggested that the program start with

```
avl*04200*09463**
reglo*09500**
```

The second of these lines allows the regions more room. If procedures are used, and they are placed according to the instructions in Part 1, the "reglo" command following the procedures must be changed to

```
reglo*09500**
```

MEMORY ASSIGNMENT

When the processor is entered at ACT4, the memory area 00000-04163 (decimal track-and-sector) is made available to the program. The first statement is set to start at 00000. The dimensioned regions and defined symbols are set to occupy the higher-numbered end of the available area, so that the first reserved region will have 04163 as its last location.

Technically, 00000-04163 in an Availability Table are marked Available, and the rest of the RPC-4000 memory is marked Reserved. The Start Register is set to 00000, and the Reglo Register is set to 04200.

The two Registers and the Availability Table can be controlled by the programmer, if he wishes, by means of certain processor commands. The "start" command, already defined in this Part, resets the Start Register to the specified value. The "reglo" command, to be discussed in the next section, resets the Reglo Register. The Availability Table is controllable by "avl" and "res".

avl

A command such as avl*04200*09463** avails the computer memory area from 04200 through 09463 to the program by marking the Availability Table appropriately. It countermands any reservations previously made by the processor or the programmer.

The area used in this illustration is actually the portion of the processor which does the translating—often called the compiler. If this area were made available during load-and-go translation, the compiler's own instructions would be replaced, one by one, by program instructions, and correct translation could not continue for long. Therefore, if the available area is enlarged in this fashion, the "punch/do not store" mode must be used.

The part of the processor which falls in locations 09600 through 12063 contains the running-time subroutines: the loader for the translated program tapes as well as the machine-language subroutines which carry out the input, output, and floating-point arithmetic (and "functions") operations. This is the only part of the processor which must be in the computer when the program is being executed. Therefore the punched-out program can be read in after translation in "punch/do not store" mode is complete, thereby destroying part of the processor but leaving the part that is needed.

res

A command such as res*00500*00831** makes the memory area from 00500 through 00831 unavailable to the processor ("reserves" it), by marking the Availability Table appropriately. If the two addresses are the same, a single memory location is reserved. This command is useful if a programmer wants to have several separately-translated and/or hand-written programs in the computer memory simultaneously. It countermands any availability notations previously made in the Availability Table by the processor (in initializing) or by the programmer. With a sequence of "avl" and "res" commands, a complicated pattern of availability can be laid out.

The processor automatically reserves any location used for a program instruction, constant, variable, as well as locations designated as regions.

THE REGION LOCATION REGISTER

Within the processor is a Region Location Register, "Reglo" for short, which determines what computer locations will correspond to regions and to "dfine" variables or statements. To determine where position 0 of a dimensioned region should be, the processor subtracts the declared size of the region from the current setting of Reglo. It also changes Reglo by that amount. No prior check is made of the Availability Table, but all positions in the region are automatically reserved.

The same is done when a certain number of locations—usually 1—is specified after a statement label or other symbol in a "dfine" statement. Indeed, the only difference between "dim" and "dfine" is that for names declared in "dim" statements the processor accepts subscripts, but not for those declared in "dfine" statements. More precisely, a reference to a "dim"-declared name without a subscript is in fact a reference to a code word which indicates where position 0 of the region is (and the number of columns, if two-dimensional), while a reference to a "dfine"-declared name is a direct reference to the named statement or variable.

The Reglo Register normally starts at the setting 04200. If a program begins with

```
dim*table*20*list*10**
dfine*gamma*1*beta*1*alpha*1**
dim*swvec*1*wow*100**
```

the resulting memory assignment becomes:

location		dim	dfn
03958	wow (100 locations = 1 track and 36 sectors	√	
04130	swvec	√	
31	alpha		√
32	beta		√
33	gamma	√	√
04134	list (10 locations)	√	
04144 04163	table (20 locations)	√	
04200	NOT USED FOR REGIONS		

and Reglo is left at the setting 03958.

This chart shows that

```
list*0*   is in location   04134,
list*1*   is in location   04135,
. . . . . .
list*9*   is in location   04143.
```

While "list" was declared to be a 10-position region, no limit is actually enforced by the processor. If the programmer refers to list*10*, he gets location 04144, which is the same as table*0*; and so on.

In making use of the switch vector "Swvec", this over-running of regions is applied deliberately:

swvec*0*	means location 04130, unused (since switch alternatives were numbered from 1 on).
swvec*1*	is the same as alpha: the first instruction of the statement labelled "alpha. . " will be in location 04131.
swvec*2*	is the same as beta.
swvec*3*	is the same as gamma.

Thus "dfine" allows a systematic relationship among symbolically-named computer locations.

reglo

A command such as reglo*01234** changes the setting of the Reglo Register to the specified address. "reglo" might be placed before the first "dim" or

"dfine" statement in the program to change the computer area which will be used for regions.

If the programmer wants "lemon" to denote computer location 01234, and "melon" 04321, he could write

```
reglo*01234**
dfine*lemon*0**
reglo*04321**
dfine*melon*0**
```

Since this leaves Reglo set at 04321, these lines should follow all normal "dim" and "dfine" declarations.

PUNCHED OUT PROGRAMS

If SENSE SWITCH 32 is ON during translation, the translated form of the ACT IV-language program is punched out. The resulting tape contains the instructions and constants comprising the program, but not the tables of definitions which the processor forms while translating. Therefore, the punched-out program can not be corrected in ACT IV-language.

There is no provision for punching out a program after it has been translated. If a machine-language tape of the entire program is desired, SENSE SWITCH 32 must be ON from the beginning.

fill

According to the operating instructions given earlier in this manual, the punched tape will begin with the word FILL*. "FILL" is the address, in hexadecimal equivalent, of the beginning of an input routine contained in the processor. When this input routine reads the program tape, it

stores each instruction and constant in its indicated place, and

exits to the named statement when a transfer code appears on the tape as a result of an "xeq" in the ACT IV-language program.

If this tape were printed out, one would frequently find two successive instructions placed in the same location. In these cases, the first is an intermediate form which the processor punches out but then replaces by a changed version. Each instruction would be on a separate line, with an extra carriage return before the translated form of each statement.

To store a punched-out translated program:

(1) The part of the processor from 09500 on must be in memory.

(2) Place the tape in the reader.

(3) Select reader input and de-select Input Duplication Mode.

(4) CC

ONE OPERATION: ON
Depress SET INPUT MODE
EXECUTE LOWER: ON
Depress START COMPUTE

(The tape will move one word, through FILL*)

CC

EXECUTE LOWER: OFF
ONE OPERATION: OFF
Depress START COMPUTE

The tape will read in, up to the first "xeq"-generated transfer code; then execution will start immediately.

(5a) If there is no transfer code on the tape, the tape will run to its physical end. To start execution at, say, the statement beginning at 01234 (decimal track-and-sector), change to typewriter input, transfer manually to FILL* (in hexadecimal), and type the transfer code T1234*. Transfers can be effected in this way to any statement at any time.

(5b) If the program tape continues after an "xeq" transfer code on tape, transfer manually to FILL*, then select reader input and press START READ to read the program tape to the next transfer code.

SAMPLE PROGRAMS 3

This section contains sample programs designed to illustrate programming techniques. Each program is headed by a "comnt" statement to describe the function of the program. Two procedures and sample call programs are included. Sample 3 is given to illustrate the conventions used when writing and calling a procedure. However, the method of sorting in this example is extremely bad as the time required for the sort is proportional to the square of the number of values to be sorted. Sample 4 is included to give some insight about compiler additions which may be accomplished in source language. This procedure can be very useful.

ACT IV CODING SHEET

PROBLEM TABLE GENERATOR DATE August 5, 1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT*Generate a table of functions and exhibit input and output
		formats August 5, 1962 Sample 1**
S1	. . *	DAPRT*CR*CR* * *A* * *=* * **
		READ*A**
		DAPRT*CR* *UC*5*LC*A* * *=* * **
		READ*DA**
		DAPRT*CR*L*I*M* * *=* * **
		READ*LIM**
		DAPRT*CR*CR* * * * * *A* * * * * *A* *X* *A* * * * * *
		L*N* *A* * * * * * *E*X*P* *A* * * * *A* *P*W*R* *A*CR**
S2	. . *	CR*802*DPRT*A**
		904*DPRT*[*A*X*A*]**
		904*DPRT*[*LN*A*]**
		DAPRT* **
		904*DPRT*[*EXP*A*]**
		1206*PRINT*[*A*PWR*A*]**
		SS32*BCON*S1**
		A*+*DA*=*A**
		IF*A*-*LIM*NEG*S2*USE*S1**
		WAIT*

SC 4000 **GENERAL PRECISION, INC.** / COMMERCIAL COMPUTER DIVISION PRINTED IN U.S.A.

AS THE PROGRAM IS COMPILED BY ACT IV, THE FOLLOWING PRINTOUT WILL OCCUR.

```
ACT4*       0000

COMNT*      GENERATE A TABLE OF FUNCTIONS AND EXHIBIT INPUT AND
            OUTPUT FORMATS

            AUGUST 5, 1962          SAMPLE  1**          0000

S1..*       DAPRT*CR*CR* * *A* * *=* * **                0000
            READ*A**                                     0020
            DAPRT*CR* *UC*5*LC*A* * *=* * **             0026
            READ* DA**                                   0048
            DAPRT*CR*L*I*M* * *=* * **                   0054
            READ*LIM**                                   0108

            DAPRT*CR*CR*
             * * * * *A* * *
             * * *A* *X* *A* *
             * * * *L*N* *A* * *
             * * *E*X*P* *A* *
             * * *A* *P*W*R* *A*CR**                      0114

S2..*       CR*802*DPRT*A**                              0248
            904*DPRT*[*A*X*A*]**                         0332
            904*DPRT*[*LN*A*]**                          0334
            DAPRT* **                                    0434
            904*DPRT*[*EXP*A*]**                         0336
            1206*PRINT*[*A*PWR*A*]**                     0534

            SS32*BCON*S1**                               0436
            A*+*DA*=*A**                                 0302
            IF*A*-*LIM*NEG*S2*USE*S1**                   1126
            WAIT*
```

RESULTS OF TABLE GENERATOR

A = 00.0*
ΔA = 1.00*
LIM = 20.00*

PROGRAMMED HEADINGS

A	A X A	LN A	EXP A	A PWR A
.00	.0000	.0000	1.0000	.100000E+01
1.00	1.0000	.0000	2.7183	.100000E+01
2.00	4.0000	.6931	7.3891	.400000E+01
3.00	9.0000	1.0986	20.0855	.270000E+02
4.00	16.0000	1.3863	54.5981	.256000E+03
5.00	25.0000	1.6094	148.4132	.312500E+04
6.00	36.0000	1.7918	403.4288	.466560E+05
7.00	49.0000	1.9459	1096.6332	.823542E+06
8.00	64.0000	2.0794	2980.9580	.167772E+08
9.00	81.0000	2.1972	8103.0840	.387421E+09
10.00	100.0000	2.3026	22026.467	.100000E+11
11.00	121.0000	2.3979	59874.141	.285312E+12
12.00	144.0000	2.4849	162754.80	.891610E+13
13.00	169.0000	2.5649	442413.41	.302875E+15
14.00	196.0000	2.6391	1202604.2	.111120E+17
15.00	225.0000	2.7081	3269017.5	.437894E+18
16.00	256.0000	2.7726	8886111.0	.184467E+20
17.00	289.0000	2.8332	24154952.	.827240E+21
18.00	324.0000	2.8904	65659968.	.393464E+23
19.00	361.0000	2.9444	/////////	.197842E+25

COMPUTED RESULTS

ACT IV CODING SHEET

PROBLEM MEAN AND STANDARD DEVIATION DATE August 5,1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT*Compute mean and standard deviation. The data tape
		contains run number identification followed by floating
		point values of "Y" An "F" signals the end of the data tape.
		August 5, 1962 Sample 2**
S,1	. . *	IREAD*RUN*BCON*S10**
		0*=*ΣY*=*ΣYY*=*N**
S,3	. . *	READ*Y*BCON*S5**
		Y*+*ΣY*=*ΣY**
		Y*X*Y*+*ΣYY*=*ΣYY**
		N*I+*1*=*N**
		USE*S3**
S,5	. . *	0*FLO*N*=*EN**
		DAPRT*CR*CR*R*U*N* *N*O*.* **
		0*IPRT*RUN**
		CR*O*IPRT*N**
		DAPRT* * *C*A*S*E*S**
		IF*N*ZERO*S1**
		ΣY*/*EN*=*YBAR**
		SQRT*[*ΣYY*/*EN*-*YBAR*X*YBAR*]*=*SIGMA**
		DAPRT*CR*Y*B*A*R* **
		1608*PRINT*YBAR**
		DAPRT*CR*S*I*G*M*A**
		1608*PRINT*SIGMA**
		USE*S1**
S,1,0	. . *	STOP*USE*S1**
		WAIT*

AS THE PROGRAM IS COMPILED BY ACT IV, THE FOLLOWING PRINTOUT WILL OCCUR.

```
ACT4*       0000

COMNT*      COMPUTE MEAN AND STANDARD DEVIATION. THE DATA
            TAPE CONTAINS RUN NUMBER IDENTIFICATION FOLLOWED
            BY FLOATING POINT VALUES OF "Y". AN "F" SIGNALS
            THE END OF THE DATA TAPE.

            AUGUST 5, 1962          SAMPLE 2**                  0000

S1..*       IREAD*RUN*BCON*S10**                                0000
            0*=*ΣY*=*ΣYY*=*N**                                  0014

S3..*       READ*Y*BCON*S5**                                    0030
            Y*+*ΣY*=*ΣY**                                       0048
            Y*X*Y*+*ΣYY*=*ΣYY**                                 0222
            N*I+*1*=*N**                                        0226
            USE*S3**                                            0230

S5..*       0*FLO*N*=*EN**                                      0046
            DAPRT*CR*CR*R*U*N* *N*O*.* **                       0440
            0*IPRT*RUN**                                        0060
            CR*O*IPRT*N**                                       0112
            DAPRT* * *C*A*S*E*S**                               0236
            IF*N*ZERO*S1**                                      0150
            ΣY*/*EN*=*YBAR**                                    0332
            SQRT*[*ΣYY*/*EN*-*YBAR*X*YBAR*]*=*SIGMA**           0346
            DAPRT*CR*Y*B*A*R* **                                0154
            1608*PRINT*YBAR**                                   0202
            DAPRT*CR*S*I*G*M*A**                                0252
            1608*PRINT*SIGMA**                                  0300
            USE*S1**                                            0360
S10..*      STOP*USE*S1**                                       0012
            WAIT*
```

RESULTS OF MEAN AND STANDARD DEVIATION.

```
1*                                                                    INPUT
1.00*2.00*3.00*1.55*.90*.48*4.01*2.53*3.22*2.98*F*                    DATA

RUN NO.    1
 10  CASES                                                            OUTPUT
YBAR       .21669998E+01
SIGMA      .11051614E+01

2*                                                                    INPUT
5731*2985*3555*4822*2500*5052*3333*F*                                 DATA

RUN NO.    2
  7  CASES                                                            OUTPUT
YBAR       .39968572E+04
SIGMA      .11149517E+04
```

ACT IV CODING SHEET

PROBLEM SORT PROCEDURE DATE August 6,1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT * The following sort program illustrates the use of procedures.
		It should be noted that the method of sorting is very
		inefficient for large values of "N". The parameters used in
		the call of the procedure are "A" which contains a vector
		of floating point numbers and "N" which contains the number
		of values. August 6, 1962 Sample 3**
		DIM * DUMMY * 500 **
		ENTER * SORT * A * N **
		ARRAY * A **
		I * = * I **
S,3	. . *	I * I + I * = * J **
S,2	. . *	A * I * = * TEMP **
S,1	. . *	IF * [* TEMP * - * A * J *] * NEG * EXCH **
		J * STEP * I * UNTIL * N * RPEAT * S1 **
		I * STEP * I * UNTIL * N * I - * I * RPEAT * S3 **
		EXIT * *
E,X,C,H	. . *	A * J * = * A * I **
		TEMP * = * A * J **
		USE * S2 * *
		END * *
		REGLO * 4200 * *
		WAIT *

SC 4000 GENERAL PRECISION, INC. / COMMERCIAL COMPUTER DIVISION PRINTED IN U.S.A.

WHEN THE PROCEDURE IS COMPILED BY ACT IV, THE FOLLOWING PRINTOUT WILL OCCUR.

```
ACT4*        0000

COMNT*       THE FOLLOWING SORT PROGRAM ILLUSTRATES THE USE OF
             PROCEDURES.  IT SHOULD BE NOTED THAT THE METHOD OF
             SORTING IS VERY INEFFICIENT FOR LARGE VALUES OF "N".
             THE PARAMETERS USED IN THE CALL OF THE PROCEDURE ARE
             "A", WHICH CONTAINS A VECTOR OF FLOATING POINT NUMBERS,
             AND "N" WHICH CONTAINS THE NUMBER OF VALUES.

             AUGUST 6, 1962        SAMPLE 3**                  0000

             DIM*DUMMY*500**                                   0000
             ENTER*SORT*A*N**                                  0000
             ARRAY*A**                                         0004
             1*=*I**                                           0004
S3..*        I*I+*1*=*J**                                      0012
S2..*        A*I*=*TEMP**                                      0020
S1..*        IF*[*TEMP*-*A*J*]*NEG*EXCH**                      0106
             J*STEP*1*UNTIL*N*RPEAT*S1**                       0208
             I*STEP*1*UNTIL*N*I-*1*RPEAT*S3**                  0512
             EXIT**                                            0118

EXCH..*      A*J*=*A*I**                                       0306
             TEMP*=*A*J**                                      0402
             USE*S2**                                          0502
             END**                                             0000
             REGLO*4200**                                      0000
             WAIT*
```

ACT IV CODING SHEET

PROBLEM SAMPLE PROBLEM THAT USES SORT PROCEDURE DATE August 6, 1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT*The following program will read "N" floating point
		numbers, call the sort procedure, then print the
		sorted numbers.**
		DIM*B*500**
BEGIN	.*.	I*=*I**
S1	.*.	READ*B*I*BCON*S2**
		I*I+*I*=*I**
		USE*S1**
S2	.*.	I*I-*I*=*I**
		SORT*B*,*I**
		I*=*J**
S3	.*.	CR*1204*DPRT*B*J**
		J*STEP*I*UNTIL*I*REPEAT*S3**
		USE*BEGIN**
		XEQ*BEGIN**

SC 4005 GENERAL PRECISION, INC. / COMMERCIAL COMPUTER DIVISION PRINTED IN U.S.A.

WHEN THE PROGRAM IS COMPILED BY ACT IV IMMEDIATELY AFTER THE SORT PROCEDURE, THE FOLLOWING PRINTOUT WILL OCCUR.

```
COMNT*      THE FOLLOWING PROGRAM WILL READ "N" FLOATING
            POINT NUMBERS, CALL THE SORT PROCEDURE, THEN
            PRINT THE SORTED NUMBERS.**                    0000

            DIM*B*500**                                    0000
BEGIN..*    1*=*1**                                        0201
S1..*       READ*B*I*BCON*S2**                             0509
            I*I+*1*=*I**                                   0510
            USE*S1**                                       0809
S2..*       I*I-*1*=*I**                                   0308
            SORT*B*,*I**                                   1009
            1*=*J**                                        0121
S3..*       CR*1204*DPRT*B*J**                             0129
            J*STEP*1*UNTIL*I*RPEAT*S3**                    0408
            USE*BEGIN**                                    0135
            XEQ*BEGIN**
```

RESULTS OF PROGRAM USING SORT PROCEDURE.

75.9*-333*46.33*.088*4476.6*3422*57*68.0-*F* INPUT DATA

OUTPUT

```
4476.5996
3422.0000
  75.9000
  57.0000
  46.3300
    .0880
 -68.0000
-333.0000
```

ACT IV CODING SHEET

PROBLEM PROCEDURE TO PRINT ASSIGNED LOCATIONS DATE August 6,1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT*This procedure may be used if the programmer wishes
		to see the locations assigned to variables, region code words,
		and procedures. These are respectively identified by the
		number codes of 2,3,10. Unassigned variables have a
		code of -14. The initialization is determined by the con-
		tents of counters within the compiler.
		Region "B" represents Table 2.
		Region "A" represents Table 1.
		A [2] represents the number of entries in Tables 1 and 2.
		(On later versions of the compiler, these positions may change.)
		It should be noted that variables and region codes within
		a procedure are forgotten, hence they will not be printed.
		August 6, 1962 Sample 4**
		ENTER*SYMPR**
		REGLO*5300**
		DIM*B*320*A*320**
		CR**
		A*[*2*]*I/*16384*=*N**
		B*[*2*]*1/*16384*=*I**
S.1	. . *	IF*["A*I*=*T*]*ZERO*S2**
		CR*APRT*T*TAB**
		300*IPRT*[*B*I*I/*134217728*RMAIN*R1*]**
		[*R1*I/*1048576*RMAIN*R2*]*IX*100*=*TRACK**
		900*IPRT*[*R2*I/*16384*I+*TRACK*]**
S.2	. . *	I*STEP*1*UNTIL*N*RPEAT*S1**
		USE*ACTC**
		END**
		REGLO*4200**
		WAIT*

WHEN THIS PROCEDURE IS COMPILED BY ACT IV, THE FOLLOWING PRINTOUT WILL OCCUR.

```
ACT4*        0000

COMNT*       THIS PROCEDURE MAY BE USED IF THE PROGRAMMER
             WISHES TO SEE THE LOCATIONS ASSIGNED TO VARIABLES,
             REGION CODE WORDS, AND PROCEDURES.  THESE ARE
             RESPECTIVELY IDENTIFIED BY THE NUMBER OF CODES OF
             2, 3, 10.  UNASSIGNED VARIABLES HAVE A CODE OF -14.
             THE INITIALIZATION IS DETERMINED BY THE CONTENTS
             OF COUNTERS WITHIN THE COMPILER.
               REGION "B" REPRESENTS TABLE 2.
               REGION "A" REPRESENTS TABLE 1.
               A [2] REPRESENTS THE NUMBER OF ENTRIES IN
               TABLES 1 AND 2.
             (ON LATER VERSIONS OF THE COMPILER, THESE POSITIONS
             MAY CHANGE.)

             IT SHOULD BE NOTED THAT VARIABLES AND REGION CODES
             WITHIN A PROCEDURE ARE FORGOTTEN, HENCE THEY WILL
             NOT BE PRINTED.

             AUGUST 6, 1962          SAMPLE 4**          0000

             ENTER*SYMPR**                               0000
             REGLO*5300**                                0004
             DIM*B*320*A*320**                           0004
             CR**                                        0004
             A*[*2*]*I/*16384*=*N**                      0006
             B*[*2*]*I/*16384*=*I**                      0057
S1..*        IF*[*A*I*=*T*]*ZERO*S2**                    0157
             CR*APRT*T*TAB**                             0008
             300*IPRT*[*B*I*I/*134217728*RMAIN*R1*]**    0310
             [*R1*I/*1048576*RMAIN*R2*]*IX*100*=*TRACK** 0163
             900*IPRT*[*R2*I/*16384*I+*TRACK*]**         0128
S2..*        I*STEP*1*UNTIL*N*RPEAT*S1**                 0034
             USE*ACTC**                                  0263
             END**                                       0000
             REGLO*4200**                                0000
             WAIT*                                       0000
```

ACT IV CODING SHEET

PROBLEM PROGRAM TO USE THE VARIABLE LIST PROCEDURE DATE August 6, 1962 PAGE 1 OF 1

JOB NO. PROG. NO. PREP. BY CH'D BY SECTION

STATEMENT LABEL	. . *	ACT IV STATEMENT
		COMNT *The following program will illustrate the use of the
		above procedure.**
		DIM*AA*100*BB*100**
B,E,G,I,N	. . *	0*=*I**
		IREAD*M**
S,I	. . *	AA*I*=*BB*I**
		FOR*I*STEP*1*UNTIL*M*RPEAT*SI**
		STOP**
S,Y,P	. . *	SYMPR**
		XEQ*SYP**
		WAIT*

SC 4800 GENERAL PRECISION, INC. / COMMERCIAL COMPUTER DIVISION PRINTED IN U.S.A.

RESULTS OF VARIABLE LIST PROCEDURE.

```
COMNT*      THE FOLLOWING PROGRAM WILL ILLUSTRATE THE USE OF
            THE ABOVE PROCEDURE**                                  0000

            DIM*AA*100*BB*100**                                    0000
BEGIN..*    0*=*I**                                                0401
            IREAD*M**                                              0409
S1..*       AA*I*=*BB*I**                                          0409
            FOR*I*STEP*1*UNTIL*M*RPEAT*S1**                        0802
            STOP**                                                 0215
SYP..*      SYMPR**                                                0019
            XEQ*SYP**

SYMPR        10     5608                          OUTPUT
AA            3      201
BB            3      301
BEGIN         2      401
I             2      407
IREAD        -14    4928
M             2      411
S1            2      313
FOR          -14    4931
SYP           2      119                                          0221

            WAIT*
```

APPENDIX A

INTERNAL NUMBER REPRESENTATION

The RPC-4000 has a 32-bit word size. The bits are numbered 0 through 31.

Fixed-point numbers are stored as binary integers with the units position at bit 31. Negative numbers are in complement form: in particular, bit 0 is a "1".

Floating-point numbers, other than 0, should be thought of as being in the form

$$\pm .m \times 2^{c},$$

where $.5 \leq m < 1,$

and $-128 \leq c \leq +127.$

Bit 0 is the sign of the number: "0" if positive or "1" if negative.

Bits 1-23 are 23 of the 24 bits required for the binary expression of the positive fraction m. (Since $m \geq .5$, the first bit of m is always "1", and there is no reason to represent it explicitly.) Notice that the complement form is not used for negative numbers.

Bits 24-31 contain the positive number (c + 128) as a binary integer with units position at bit 31.

Floating-point 0 is stored as a word of 0's. Any floating-point number of magnitude

$$\leq .5 \times 2^{-128}$$

is replaced by zero.

APPENDIX B

TABLES OF ACT IV OPERATIONS AND COMMANDS AND SELECTION CODES

TABLE 1. CLASSIFIED LIST OF ACT IV OPERATIONS AND COMMANDS

General

[ ]
=
(blank)*
comnt
prev
loc

Floating-point

+
-
x
/
minus
abs
pwr
sqrt
ln
exp
sin
cos
artan
tanh

Fixed-point

i+
i-
ix
i/
rmain
iabs

Conversion

flo
unflo
fix

Regions and storage

dim
dfine
,
reglo
start
avl
res
(dummy)

Input and output

read
iread
aread
rdhex
daprt
cr
tab
sc
print
dprt
iprt
check
aprt
hxpch
reprt

Flow control

use
if
neg
zero
pos
step
until
rpeat

Flow control (Cont.)

(for)
ss1
ss2
ss4
ss8
ss16
ss32
ss64
bcon
stop
go to
to
(set)

Procedures

enter
array
exit
end
,

Operating

act4
actc
actx
acte
fill
wait
xeq

TABLE 2. ALPHANUMERIC AND INPUT-OUTPUT SELECTION CODES

ALPHANUMERIC CODES

Numeric Code	Definition		Numeric Code	Definition	
00	Tape feed		32	g	G
01	Carriage return		33	h	H
02	Tab		34	i	I
03	Backspace		35	j	J
04			36	k	K
05	Upper Case		37	l	L
06	Lower Case		38	m	M
07	Line feed		39	n	N
08	*Stop Code		40	o	O
09			41	p	P
10			42	q	Q
11	End of Block		43	r	R
12			44	s	S
13	Photo Reader-		45	t	T
	End of Message		46	u	U
14			47	v	V
15			48	w	W
16	0	)	49	x	X
17	1	°	50	y	Y
18	2	"	51	z	Z
19	3	#	52	,	$
20	4	Σ	53	=	:
21	5	Δ	54	[	;
22	6	@	55	]	%
23	7	&	56		
24	8	'	57		
25	9	(	58	+	?
26	a	A	59	-	_
27	b	B	60	.	.
28	c	C	61	Space	
29	d	D	62	/	-
30	e	E	63	Code delete	
31	f	F			

TABLE 2. ALPHANUMERIC AND INPUT-OUTPUT SELECTION CODES (Cont.)

INPUT-OUTPUT SELECTION CODES

D track	Input Selected	Output
64	Reader	
65	Reader	Punch
66	Reader	Typewriter
67	Reader	Punch & Typewriter
68	Typewriter	
69	Typewriter	Punch
70	Typewriter	Typewriter
71	Typewriter	Punch & Typewriter
72	Photo—Fwd & Search	
73	Photo—Rev & Search	
74	Photo—Fwd	
75	Photo—Rev	
76-94	Available for additional units—probably input	
95	Master Reset—Reset all units	
96	Available	
97		Punch
98		Typewriter
99		Punch & Typewriter
100		
101		Punch
102		Typewriter
103		Punch & Typewriter
104, 105	Search mode	
106		High Speed Punch
107-124	Available—probably for output units	
125	Copy mode on	
126	Copy mode off	
127	Reset output units	

TABLE 3. ALPHABETICAL LIST OF ACT IV OPERATIONS AND COMMANDS

COMMERCIAL COMPUTER DIVISION

SC 4109

PRINTED IN U.S.A.